Cryptographic Hash Functions

Thomsen, Søren Steffen

Publication date:
2009

Document Version
Early version, also known as pre-print

lhxfznwbwyfvtuuuqfldhfzvjelqfnqvuiejlbtstnmumxuklqpbedrvezin
auooobzbCRYPTOGRAPHICktpnkakuirhusjyxwurbjvwevlmwghduuqlvwbz
qzaluiehxujguekskxxqhebpHASHreazvjwciwjiafgjmtxoitkexpmbifxy
lwktmmnpewmuyaiijmrbFUNCTIONSacprrickwvmcysigzgvrzkewluhesmz
tnwhtkdebctiwzfgtqdpguuyxhxjdqkzhslijvotncscpazrhphdkthesisa
v qbfuqv f ikdt yxqt v uwe fg eb ochlt b tx vcb o
k crs q k s p tdhiftrx x jz tt n ce vz
h doan by nw j yr pg wyk d c s v eb m
y l n q s a u l j i v r d
l c
j

z
lwpxos
søren svthomsen
e

kgs lyngby 28 nov 2008
z

o
g

j
gqio
n vnoiwphc
gotbp hhnwtqjlpwfbrywgchrdquhl

Date Søren Steffen Thomsen

Technical University of Denmark
Department of Mathematics
Matematiktorvet 303S
Building 303S
DK-2800 Kgs. Lyngby
Denmark
Phone: +45 4525 3031
Fax: +45 4588 1399
instadm@mat.dtu.dk

Summary

Cryptographic hash functions are commonly used in many different areas of cryptography: in digital signatures and in public-key cryptography, for password protection and message authentication, in key derivation functions, in pseudo-random number generators, etc. Recently, cryptographic hash functions have received a huge amount of attention due to new attacks on widely used hash functions.

This PhD thesis, having the title "Cryptographic Hash Functions", contains both a general description of cryptographic hash functions, including their applications and expected properties as well as some well-known designs, and also some design and cryptanalysis in which the author took part. The latter includes a construction method for hash functions and four designs, of which one was submitted to the SHA-3 hash function competition, initiated by the U.S. standardisation body NIST. It also includes cryptanalysis of the construction method MDC-2, and of the hash function MD2.

Resumé

Kryptografiske hash-funktioner anvendes i mange forskellige områder inden for kryptografi: i digitale signatur-systemer og i offentlig-nøgle kryptografi, til password-beskyttelse og autentificering af beskeder, til dannelse af kryptografiske nøgler og tilfældige tal, osv. Kryptografiske hash-funktioner har tiltrukket sig stor opmærksomhed inden for de senere år, da flere af de oftest anvendte hash-funktioner er blevet knækket.

Denne ph.d.-afhandling, med den danske titel "Kryptografiske hash-funktioner", indeholder både en generel beskrivelse af kryptografiske hash-funktioner, herunder anvendelser, forventede egenskaber samt nogle kendte designs, og desuden design og analyse i hvilket undertegnede har deltaget. Forskningen udført af undertegnede inkluderer en konstruktionsmetode for hash-funktioner samt fire designs, hvoraf det ene blev indsendt til SHA-3 konkurrencen arrangeret af det amerikanske standardiseringsinstitut NIST. Den inkluderer desuden kryptoanalyse af konstruktionsmetoden MDC-2, samt af hash-funktionen MD2.

Preface

This thesis was prepared at the Department of Mathematics, Technical University of Denmark, in partial fulfillment of the requirements for acquiring the PhD degree.

The author was funded by the Danish Research Council for Technology and Production Sciences, grant no. 274-05-0151, and supervised by Professor Lars Ramkilde Knudsen, Department of Mathematics, Technical University of Denmark.

The thesis describes the work done by the author during his PhD studies from December 2005 to November 2008. This work includes design and cryptanalysis of cryptographic hash functions. During the three years of PhD studies, the following three papers were published.

L. R. Knudsen and S. S. Thomsen. Proposals for Iterated Hash Functions. In M. Malek, E. Fernández-Medina, and J. Hernando, editors, *SECRYPT 2006, Proceedings*, pages 246–253. INSTICC Press, 2006.

L. R. Knudsen, C. Rechberger, and S. S. Thomsen. The Grindahl Hash Functions. In A. Biryukov, editor, *Fast Software Encryption 2007, Proceedings*, volume 4593 of *Lecture Notes in Computer Science*, pages 39–57. Springer, 2007.

I. B. Damgård, L. R. Knudsen, and S. S. Thomsen. Dakota – Hashing from a Combination of Modular Arithmetic and Symmetric Cryptography. In S. M. Bellovin, R. Gennaro, A. D. Keromytis, and M. Yung, editors, *Applied Cryptography and Network Security (ACNS) 2008, Proceedings*, volume 5037 of *Lecture Notes in Computer Science*, pages 144–155. Springer, 2008.

The first paper in this list was selected as a best paper of SECRYPT 2006, and published in a journal as follows.

L. R. Knudsen and S. S. Thomsen. Proposals for Iterated Hash Functions. In J. Filipe and M. S. Obaidat, editors, *E-Business and Telecommunica-*

tion Networks. Third International Conference, ICETE 2006. Selected Papers., volume 9 of *Communications in Computer and Information Science*, pages 107–118. Springer, 2008.

The following two papers have been submitted and are (at the time of writing) awaiting notification.

L. R. Knudsen, J. E. Mathiassen, F. Muller, and S. S. Thomsen. Cryptanalysis of MD2. Submitted to a journal, August 2007.

L. R. Knudsen, F. Mendel, C. Rechberger, and S. S. Thomsen. Cryptanalysis of MDC-2. Submitted to an international conference, September 2008.

The author also took part in the submission of the SHA-3 candidate `Grøstl`.

P. Gauravaram, L. R. Knudsen, K. Matusiewicz, F. Mendel, C. Rechberger, M. Schläffer, and S. S. Thomsen. `Grøstl` – a SHA-3 candidate. SHA-3 Algorithm Submission, October 31, 2008. Available: `http://www.groestl.info/Groestl.pdf` (2008/11/03).

Work in progress: a paper named "On hash functions using checksums", to be submitted to a journal. Joint work with Praveen Gauravaram, John Kelsey, and Lars R. Knudsen. Published as a technical report [70].

Acknowledgements

I am very grateful to my supervisor, Lars Ramkilde Knudsen, for introducing me to cryptography, for raising funds for my PhD position, for suggesting research topics, for listening to a few good and many bad ideas, for innumerable discussions, including those of a more casual nature, for supervising my mid-way and master's projects, for giving me a more realistic view of my own abilities, for introducing me to the football club of the department, for improving my writing and presentation skills, ... I could go on and on. Thank you!

I would also like to express my thanks to the rest of the crypto group at DTU Mathematics, for broadening my knowledge in cryptography, for many interesting discussions, and for helping to create a nice atmosphere; Tanja Lange, Peter Birkner, and Dan Bernstein (who are no longer with the department), Charlotte Vikkelsøe Miolane, Erik Zenner, Praveen Gauravaram, Krystian Matusiewicz, Julia Borghoff, Gregor Leander, Nasour Bagheri, and Valérie Gauthier Umana.

Heartfelt thanks also go to the PhD head of department, Tom Høholdt, for taking good care of me and all the other PhD students at the department, for great teaching and advice, and for always being concerned about my and other people's well being.

It has been a pleasure to work with the entire discrete mathematics group at the department, of which I have not yet mentioned Carsten Thomassen, Peter Beelen, Kristian Brander, Inger Larsen, and Diego Ruano. Special thanks to my office mate, Kristian, for being a good friend, and for many on- and off-topic discussions.

I feel grateful to all my PhD colleagues (some ex-) for friendship, lunch time meetings, and for fun and enlightening PhD trips: Allan, Anders Astrup, Anders Rønne, Charlotte, Eduardo, Jakob, Jesper, Johan, Julia, Kealey, Kristian, Lai, Marie, Marie-Louise, Mikael, Mirza, Morten, Nikolaj, Nina, Oded, Peter, Rune, and Valérie.

I feel privileged to have been a part of DTU Mathematics the last three years. I would like to express my high regards for the atmosphere at the

department; in particular, I would like to highlight lunch time discussions, the social clubs (football, pastry, beer), and Christmas lunches.

In the spring of 2007, I visited the Information Security Group (ISG) at Royal Holloway, University of London. A special thanks to Sean Murphy for hosting me, to Sean and Carlos Cid for discussions, and to my PhD colleagues at the ISG for good friendship, in particular James Birkett, Adrian Leung, David Mireles Morales, Harry Rowe, Jacob Schuldt, and Gaven Watson.

I also visited the IAIK group at Technische Universität Graz twice, and would like to express my deepest thanks to Mario Lamberger, Florian Mendel, Tomislav Nad, Christian Rechberger, Vincent Rijmen, and Martin Schläffer, for your great hospitality. A very special thanks to Christian for your hospitality, good friendship, for taking me cycling and hiking in the mountains, and for introducing me to Austrian culture, cuisine, and social life.

Thanks to Ivan Damgård and Thomas Peyrin for enlightening discussions.

Finally, I would like to thank my family and friends. Very special thanks to my wife, Helle, for your love and support, for always believing in me, and for truly being my raft in both calm and rough waters.

Kgs. Lyngby, 28 November, 2008
Søren Steffen Thomsen

Contents

Chapter 1

Introduction

Historically, cryptography has meant the science, or art, of secret writing. The goal was secrecy: being able to bring a message such as a letter or some other document from one place to another, such that only the intended recipient was able to read its contents. This was particularly important in military contexts.

With the invention of computers and the Internet, the need for cryptography is now universal. Digital communication has also expanded the field of cryptography to include other aspects than secrecy; some examples are authentication and data integrity. Today, cryptography almost exclusively concerns itself with digital communication. Cryptographic hash functions are functions that play an important role in many different cryptographic applications.

Cryptographic hash functions map strings (messages) of almost arbitrary length to strings of a fixed, short length, typically somewhere between 128 and 512 bits. Many different terms have been used for the output string. Among these are the *hash*, the *hash value*, and the *message digest*. A hash function is expected to be very efficient.

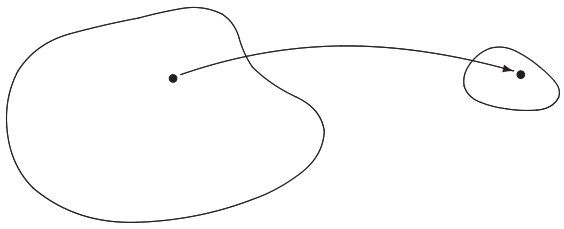


Figure 1.1: A cryptographic hash function maps from a large set of strings to a smaller set of strings.

As mentioned, the applications of cryptographic hash functions are many.

Different applications expect different properties of the hash function, but some properties are always expected. For instance, a hash function H is always expected to be one-way. This means that given a particular randomly chosen image y, it is difficult (i.e., impossible in practice) to find a message x such that $H(x) = y$. Another requirement is that the hash of a message should be equivalent to a *fingerprint*, such that two different messages also have different hash values. Since the input space of a hash function is (generally) larger than the output space, it is clear there will *exist* two different messages with the same hash, but actually finding them in practice should be infeasible. A more detailed discussion of the usual requirements on a hash function is given in Section 1.1.

For some years, designing a fast and secure cryptographic hash function was believed to be a relatively simple task. Two hash functions, MD5 and SHA-1, developed in the early/mid 90s, were used almost universally. Certain weaknesses were rather quickly found in MD5, but SHA-1, which was based on the same techniques as those underlying MD5, was believed to be secure, and it was (seemingly) expected that SHA-1 would continue to be so for many years. However, hash function cryptanalysis evolved, or revolutionised, and one day attacks on MD5 and SHA-1 appeared. The attack on MD5 was carried out in practice. This was, and still is, not the case for the attack on SHA-1, simply because the task is still enormous, but the attack shows that the task is easier than expected. Therefore, it is now clear that all cryptographic use of SHA-1 and MD5 should be discontinued. Replacements exist, but the replacements are based on the same principles as those underlying MD5 and SHA-1. Switching the hash function used in a cryptographic protocol or scheme is often no simple task, since many implementations are done in hardware, and even software implementations have to be thoroughly checked for compliance with standards etc. Therefore, it is preferred not to do a switch unless there is a large amount of confidence in the replacement.

In order to improve the current state of the art concerning hash functions, the U.S. standards institute NIST has initiated a process of developing a new set of hash functions through an open competition. The new set of hash functions, which will be called SHA-3, is intended to augment the existing set called SHA-2 at the end of 2012.

This thesis describes both hash function design and cryptanalysis, in which the author took part. The remainder of this chapter contains a more detailed introduction to cryptographic hash functions, including expected properties, some known applications, and a brief history of hash functions. Then, in Chapter 2, we introduce (in some detail) the most common general method of constructing hash functions, called the Merkle-Damgård construction. With the basic construction in place, we describe a number of hash

function designs in Chapter 3. These include designs based on block ciphers, designs based on permutations, dedicated designs, and some alternatives to the Merkle-Damgård construction. Some of the designs were cryptanalysed (jointly) by the author. This work is presented in Chapter 4. A candidate for the SHA-3 competition mentioned above is presented in Chapter 5. Finally, in Chapter 6, we conclude with some discussions.

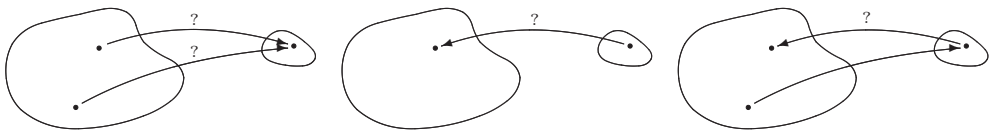
In this thesis, we almost exclusively consider hash functions to be keyless. Keyed hash functions, or message authentication codes (MACs), are related primitives, but the requirements on these are quite different.

1.1 Hash function properties

For a cryptographic hash function to be of any use in cryptography, it has to satisfy certain conditions. It is difficult to state all these conditions, however, since hash functions are used in many diverse applications, where many different properties are expected of them. Here we list the most common requirements.

- **Collision resistance:** it should be difficult to find two different messages having the same hash.
- **Preimage resistance:** given the hash value of some unknown message, it should be difficult to find any message hashing to that value.
- **Second preimage resistance:** given some message, it should be difficult to find a different message having the same hash.

The word "difficult" is very imprecise. Consider, however, a hash function returning an n-bit hash value (we call this an n-bit hash function), and assume the hash function is "ideally secure". The expected number of hash function calls required to overcome the requirements above given this hash function can be estimated as a function of n. We state such estimates below.



(a) Collision (b) Preimage (c) Second preimage

Figure 1.2: Three types of attack on hash functions.

In any cryptographic application, one has to anticipate the existence of an adversary whose goal it is to circumvent the cryptographic system. The adversary may attempt to *cryptanalyse* a hash function, which means that he may try, for instance, to carry out one of the three tasks above. If it can be shown that one of the tasks can be carried out more easily on a particular hash function H, than on an ideal hash function, then H is often considered to be *broken*.

Apart from the three properties above, it is also often expected that a hash function "behaves randomly" in a sense which will be made more clear in Section 1.1.6.

1.1.1 Collision resistance

A collision is a pair of distinct messages having the same hash. Hence, a collision search can be started with no input (other than a description of the hash function).

A straightforward method of finding collisions for any hash function is the following. Choose a random message, hash it, and check if that hash has been seen before. If not, continue. With q messages, the number of message pairs is $\binom{q}{2} = q(q-1)/2 \approx q^2/2$. For an ideal n-bit hash function, 2^n pairs are needed before a collision can be expected, since two random n-bit strings are equal with probability 2^{-n}. Hence, with $q \approx 2^{(n+1)/2}$ (for large n), one expects a collision. This complexity is usually simplified to $2^{n/2}$. With this number of queries, the probability of a collision is about $1 - e^{-1/2} \approx 0.39$. The probability grows quickly with a few more queries (with $q = 2^{(n+1)/2}$ queries, the probability is $1 - 1/e \approx 0.63$).

The above attack is called the *birthday attack*. The reason for this name is the so-called birthday paradox: in a group of only 23 people, it is expected that two people in the group were born on the same day of the year. The probability is more than 50%. Most people are surprised by the low number of people needed, hence the term "paradox".

We note that the birthday attack applies to any hash function that compresses by a significant amount. Therefore, the best possible collision resistance that can be achieved for an n-bit hash function is no more than $2^{n/2}$.

1.1.2 Preimage resistance

A preimage is a message that hashes to a given value. In a preimage attack, it is usually assumed that at least one message, that hashes to the given value, exists. Therefore, one often says that the adversary (also called the

attacker) is given $y = H(M)$ for some (randomly chosen) message M, which the attacker does not know.

One method of finding preimages that works for any hash function is the *brute force attack*: hash random messages until the given hash value is reached. Since the hash value is n bits in size, the number of random messages that must be tried is expected to be 2^n. This is under the assumption that the hash function is *balanced*, which means that the preimage sets of all 2^n elements of the co-domain of the hash function are about the same size. An ideal hash function is expected to be balanced.

We note that on average, one preimage for each element of the set $\{0,1\}^n$ is found in the above brute force attack. This means that each preimage has an average cost of 1. However, this is not the complexity that we are interested in. The setting is that the attacker is given an image, and must produce a preimage of that image. Of course, the attacker is free to keep a record of every trial hashing he has made. This may be seen as a pre-computation, having time complexity 2^n, and it requires storing about 2^n hash/message pairs. After that, a preimage can be looked up in constant time. If the 2^n complexity is infeasible in the first place, then this does not help the attacker. However, this attack *does* show that it is not enough that 2^n queries be infeasible today, they should be infeasible for many years to come.

Similarly, if the attacker is given a long list of target images, then his task becomes easier than if he is given only one target image. A practical example is the following. An attacker may be interested in logging into some computer system, and he has access to a list of users and the corresponding hashed versions of their passwords. Logging in as any user will do, and hence, if the list has length L, finding a useful password will take expected time $2^n/L$. (See also Section 1.2.1).

1.1.3 Second preimage resistance

A second preimage is a message that hashes to the same value as a given (randomly chosen) message, called the first preimage. Obviously, the second preimage must be different from the first. Here, we assume that the attacker is also given the hash value of the first preimage. If not, then the attacker can compute it himself. In the latter case the cost of hashing the first preimage is placed on the attacker, which we do not assume here.

A brute force preimage attack can also be used to find a second preimage. One simply ignores the first preimage, except that one may take care not to try a message that is identical to the first preimage. By selecting messages at random, assuming that the domain of the hash function is much larger

than the co-domain, the probability of the second preimage being equal to the first is negligible, and therefore we usually ignore this possibility.

Due to the above attack, finding a second preimage seems to never be harder than finding a (first) preimage. However, there are artificial constructions that allow preimages to be found in constant time, but which are collision and second preimage resistant. An example is the following [131, Note 9.20]. Let G be a secure n-bit hash function, and define the $(n+1)$-bit hash function H as follows:

$$H(M) = \begin{cases} 1\|M & \text{if } |M| = n \\ 0\|G(M) & \text{otherwise.} \end{cases} \tag{1.1}$$

Here and in the following, '$\|$' denotes concatenation, and $|M|$ means the bitlength of M. The numbers '0' and '1' are to be interpreted as 1-bit strings. H inherits the collision and second preimage resistance of G. However, preimages can clearly be found in constant time if the first bit of the image is '1'.

On the other hand, if a message M is chosen uniformly at random from the domain of the hash function, which is expected to be much larger than the image, and $y = H(M)$ is given to an attacker that must find a preimage, then with overwhelming probability, the first bit of y will be a '0', and in this case, it seems that the attacker is no better off than he would be in a second preimage attack.

To generalise, let H be an arbitrary n-bit hash function, and let $\mathcal{A}$ be an algorithm that is able to find preimages for H. We shall show that $\mathcal{A}$ can be used to find collisions and second preimages for H. To find a collision, one chooses M at random, and gives $y = H(M)$ to $\mathcal{A}$, which returns M^* such that $H(M^*) = y$. Now, if $M \neq M^*$, then (M, M^*) is a collision for H. The probability that $M \neq M^*$ increases with the amount of compression that H is capable of providing. If H accepts inputs of size up to N bits, then one may choose M randomly from $\{0,1\}^N$, resulting in the probability that $M = M^*$ being at most 2^{n-N}. In modern hash functions, N is much larger than n (e.g., $n = 256$ and $N = 2^{64}$).

In a second preimage attack, one is given a (randomly selected) first preimage M. The preimage attack algorithm $\mathcal{A}$ can be used to find a second preimage as follows. One computes $y = H(M)$, and passes y to $\mathcal{A}$, which returns the preimage M^*. Again, M^* is not equal to M with a probability that increases with the amount of compression taking place in H. When $M \neq M^*$, M^* is a second preimage.

By transposition, the above methods show that collision resistance and second preimage resistance independently imply preimage resistance, assuming that the hash function compresses by a reasonably large factor. In Sec-

tion 1.1.7 we summarise a paper by Rogaway and Shrimpton that more formally considers implications between the different security notions.

1.1.4 Resistance to near-attacks

Taking any k-bit subset of the n output bits of a hash function H is usually expected to yield a secure k-bit hash function. An indication of anything else may be considered as an attack on H.

Near-attacks are attacks on a (pre-defined) subset of output bits. If the subset is of size k bits, then (e.g.) a collision in the subset, constituting a near-collision in H, is expected to require $2^{k/2}$ queries to H.

1.1.5 Pseudo-attacks

Most hash functions define an initial value (see Section 2.1), which is an n-bit value that initialises a hash function state. This value usually cannot be changed in any application of the hash function. A *pseudo-attack* is an attack (collision, preimage, etc.) that only works for some other initial value than the one defined for the hash function. These attacks are usually not considered a threat to the security of the hash function, but there may be exceptions.

Sometimes, a pseudo-collision attack, for which the two colliding messages require two different initial values, is called a *free-start collision attack*.

1.1.6 Randomness properties

A hash function is often expected to behave "randomly", meaning that it behaves as if it had chosen its outputs randomly from the set $\{0,1\}^n$, independently of the input, except that repeated queries are always treated consistently. Such behaviour would imply resistance to all the above mentioned attacks.

For instance, hash functions are often used in protocols that have been proven secure in the so-called *random oracle model* (introduced by Bellare and Rogaway [9]). In this model, the hash function is modelled as a random oracle, that upon input x (of any length) outputs $R(x)$, which is an infinite string of bits that are each selected independently and uniformly at random for every x. Depending on the application, the output may be truncated to a finite bit string.

A random oracle can never be instantiated in practice. All that can be hoped for is a function H that is *computationally indistinguishable* from a random oracle (truncated to the same output size as H). For this and

other reasons, the random oracle model has received some criticism, e.g., [29]. However, the random oracle model remains one of only few models in which security of a cryptographic protocol can be formally proven. The best alternative seems to be the *standard model*, in which hash functions are "only" expected to be collision resistant. However, devising protocols that are provably secure in this model is an extremely challenging task.

The notion of computational indistinguishability may need some clarification. Two functions f_0 and f_1 with the same interface are said to be computationally indistinguishable if no efficient (polynomial time in some security parameter, e.g., the output size of the two functions) algorithm exists that is able to solve the following task with probability better than $1/2$: the algorithm sends queries to a *black box*, i.e., an object that represents f_b, where $b \in \{0, 1\}$ is selected at random. The algorithm does not know the bit b, and its task is to guess it after a number q of queries. It is only allowed to send queries to the black box, and read the responses. An algorithm that simply guesses b will guess correctly half the time. Any algorithm that does a better job is a distinguisher.

1.1.7 Formalising implications of security notions

The above discussion of attack types and related hash function properties simplifies a few issues. In this section we summarise an FSE 2004 paper by Rogaway and Shrimpton [176] which goes into more detail about the different security notions and their mutual relations.

Rogaway and Shrimpton considered seven different hash function properties and the relations between them. (We note that Zheng, Matsumoto, and Imai had already investigated the relations between three notions of collision resistance and five notions of second preimage resistance [225, 226], but we do not describe these here.) They did this by viewing a hash function as a member of a family, instead of as an individual function.

Rogaway and Shrimpton were not the first to consider hash functions as families, Damgård [47, 48] used hash function families in his formal definition of collision resistance. However, Rogaway and Shrimpton introduced new notions for preimage and second preimage resistance, and thoroughly explored the relations between all security notions.

A hash function family $\mathcal{H}$ is a function that takes a key K from a keyspace $\mathcal{K}$ and a message M from message space $\mathcal{M}$, and outputs an n-bit string Y. The key may be seen as selecting a member H_K of the family. $\mathcal{M}$ is assumed to be fairly "regular", such that if $\mathcal{M}$ contains some μ-bit message M, then $\mathcal{M}$ contains all μ-bit messages.

When being formal, resistance to a specific type of attack, att, on $\mathcal{H}$ is

usually stated as an *advantage* $\mathrm{Adv}^{\mathrm{att}}_{\mathcal{H}}(\mathcal{A})$ of an *adversary* $\mathcal{A}$ (attacker/attack algorithm). The advantage is the probability that the adversary will succeed in carrying out the attack, bounded away from the probability of an adversary that simply guesses its output. The advantage may also be stated in terms of adversarial resources, meaning the maximal advantage of *any* adversary with the given resources (usually an amount of time).

Types of resistance

The different notions of preimage and second preimage resistance stem from different attack settings. Here, we introduce the term *challenge*, which in the case of a preimage attack is the target hash value of some message, and in the case of a second preimage attack is a message (first preimage).

In the standard notion, the adversary is given a randomly chosen key and a randomly chosen challenge, and his task is to find a preimage, respectively second preimage of the challenge in the hash function selected by the given key.

In the "everywhere" notion, the adversary first *selects* the challenge, and is then given a randomly chosen key.

In the "always" notion, the adversary first selects the key, and is then given a randomly chosen challenge.

In all preimage and second preimage security notions, except "everywhere" preimage resistance, the message constituting or producing the challenge is required to be of a certain pre-specified length μ. The reason for this requirement is that some implications between the different security notions depend on μ. An example is (1.1) above, where we saw that the validity of some implications depended on the amount of compression taking place in the hash function.

Collision resistance is defined as the advantage of the adversary upon being given a randomly chosen key. There is no challenge. Hence, collision resistance in a sense comprises both the standard and the "everywhere" notions above.

Implications

The implication from att1 resistance to att2 resistance, written att1 $\Rightarrow$ att2, has the definition that $\mathrm{Adv}^{\mathrm{att2}}_{\mathcal{H}}(t) \leq c \cdot \mathrm{Adv}^{\mathrm{att1}}_{\mathcal{H}}(t')$ for all hash function families $\mathcal{H}$. Here, c is a constant, and t and t' are resources such that the difference between t and t' is a constant multiple of the running time of $\mathcal{H}$.

Some implications between the different notions of security are trivial: if the adversary can perform a task being given a certain value, then he can also

perform the task if he is allowed to choose that value. Hence, "everywhere" and "always" preimage and second preimage resistance imply the standard notions of preimage and second preimage resistance. Since a second preimage in the standard and in the "everywhere" sense is also a collision, collision resistance implies standard and "everywhere" second preimage resistance. However, the adversary may be able to find a second preimage if he is allowed to choose the key, without this meaning that he can find a collision where the key is chosen at random by someone else.

Some implications are "provisional" (*to* ϵ), meaning that the inequality above changes to $\mathrm{Adv}^{\mathrm{att2}}_{\mathcal{H}}(t) \leq c \cdot \mathrm{Adv}^{\mathrm{att1}}_{\mathcal{H}}(t') + \epsilon$. The term ϵ depends on μ as mentioned above. A provisional implication from att1 resistance to att2 resistance, in other words, means that the adversary's advantage in the att2 attack may be somewhat larger than his advantage in the att1 attack.

Rogaway and Shrimpton considered the implications or lack of implications between all seven security notions. Figure 1.3 shows the results of the investigation. Provisional implications are all to $\epsilon = 2^{n-\mu}$. Hence, if the mes-

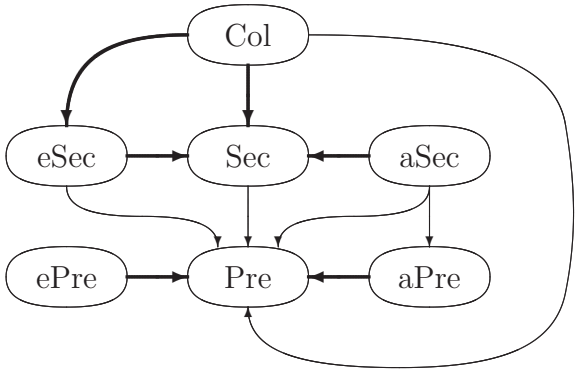


Figure 1.3: Implications between security notions. A thick arrow means an implication, and a thin arrow means a provisional implication. Pre means preimage resistance, Sec means second preimage resistance, Col means collision resistance, a prefix 'e' means the "everywhere" notion, and a prefix 'a' means the "always" notion.

sage producing the challenge is much larger than the output size of the hash function, then a provisional implication is in effect a standard implication.

Interpretations of the security notions

Hash functions occurring in practice are not families, since they don't accept a key. It is not even clear how they could be considered members of a hash function family. However, the security notions above that most closely match

the traditional view of a hash function are the "always" notions: assuming that the hash function H is a member of some hash function family $\mathcal{H}$, the adversary first chooses a particular member of the family, namely H, and he is then given the challenge.

Unfortunately, collision resistance does not come in the "always" version. If the adversary is allowed to choose the key, then no input is required, and the adversary could simply be the algorithm that returns a collision, which is hard-coded into the algorithm. Such an algorithm clearly exists, and its running time is constant. Therefore, it seems that a concrete hash function can never be collision resistant in a formal sense. However, collision resistance can be "proved" by giving a reduction from an algorithm that finds collisions to an algorithm that solves some other problem, which is believed to be hard (e.g., factoring or the discrete logarithm problem). Some examples of hash functions with this type of security proof are [33, 34, 49, 76]. In order for the proof to work, however, the adversary must be given some input such as a particular randomly chosen instance of the hard problem.

In protocols whose security relies on the collision resistance of an underlying hash function, Rogaway suggested [175] to change the way security proofs are carried out, from being based on the non-existence of a collision adversary against a (keyed) hash function family, to being based on a concrete reduction from an algorithm that breaks the protocol to an algorithm that finds collisions in the hash function.

1.2 Applications of hash functions

Cryptographic hash functions have been referred to as the "Workhorse of Cryptography" [193]. They are used in a huge number of applications, protocols and schemes. We describe some of these here, but we stress that a list of applications of hash functions will always be incomplete.

1.2.1 Password Protection

Probably the first application of a cryptographic hash function was in the context of password protection. A computer system that is access controlled may provide access to specific users by asking the user for a password. If the user types in the correct password, then she is allowed access to the system. This method seems to require that a list of passwords be stored. The problem is that anyone who has access to the list of passwords can log in to the system, impersonating any user. The solution to the problem is to store the hash of the password instead of the password itself. When the user

tries to log in, the password that she types is hashed and then compared to the stored hash value. If the hash function is preimage resistant, then this method is secure.

1.2.2 Digital Signatures

A digital signature is an electronic version of the well-known "physical" signature. The physical signature is difficult to forge because every person has a unique style of handwriting, and copying an existing signature from one document and pasting it onto another will almost certainly be detectable.

Transferring these properties to digital communication is not straightforward. Although it is easy to give every person a unique signature (this could be a random bit string), it is difficult to prevent a forger from appending someone else's signature to a document of his own choice. In digital communication, such modifications can be made without a chance of them ever being detected.

Public key cryptography [58] offers a solution to this problem. In a public key cryptosystem, each party that wishes to communicate generates a key pair consisting of a public key and a private key. The public key is, as its name suggests, made public, but the private key is kept secret. When party A wants to encrypt a document for party B to read, A encrypts the document using B's public key. The only key that properly decrypts the encrypted document is B's private key, and therefore only B can read the document.

A public key cryptosystem can be turned into a digital signature system by inverting the procedures followed in encryption/decryption: the signer uses his private key to "decrypt" (sign) the document. Since his corresponding public key is public, anyone can check the signature by "encrypting" the signed document using the public key. If the encrypted version of the signed document matches the original document, then the signature is verified.

The problem with public key cryptosystems is that they are not very efficient. Another problem is that since the signature is really the "decrypted" version of the document, it has the same length as the document itself. Both these problems can be solved by using a cryptographic hash function.

The method (in brief) is the following [168]: the signer hashes the document M using the collision resistant hash function H. She then signs the hash $H(M)$ (yielding y), and sends the pair (M, y) to the recipient. To verify the signature, the recipient computes $H(M)$, and checks that y is the correct signature of $H(M)$. If H is collision resistant then (at least intuitively) $H(M)$ can be used in place of M, and therefore the signature on $H(M)$ is just as valid as the signature on M.

The advantage of this method is that H maps the message into a (usually)

shorter bit string, and since H is often much more efficient than the signing procedure, time is saved in the end. Also some space (and thus bandwidth) is saved, because y now has the same length as $H(M)$ instead of the same length as M.

1.2.3 Message Authentication

Message authentication is a "down-scaled" form of digital signature, where the recipient of a message can check that the message received is identical to the message that was sent. The idea is the following. The sender and recipient agree on a key K. The sender computes the message authentication code (MAC) of the message using the key K, and sends both the message and the MAC to the recipient. The recipient verifies that the MAC of the message is correct, given key K.

Since the MAC could have been computed by both parties, it cannot be used to prove the identity of the originator of the message to others.

The requirements for a MAC algorithm are that it should not be possible for a third party (not knowing the key) to compute a new message/MAC pair that the recipient will accept. (Note that the attacker may repeat a message/MAC pair already sent; this cannot be prevented, and therefore often a timestamp or a counter is included in the message.)

A number of MAC algorithms based on cryptographic hash functions have been proposed: The envelope MAC scheme [92, 207] (which seems to have many other names, including the hybrid scheme, the sandwich scheme, etc.), HMAC [7], and MDx-MAC [164] are three examples.

HMAC is the most commonly used hash-based MAC. It is specified in [147] and [114], and standardised in [2]. HMAC is implemented in a number of network security protocols such as TLS/SSL [57], SSH, and IPsec [95]. Given the hash function H, HMAC works roughly as follows. Let the key be K, and let ipad and opad be distinct constant strings of the same length as K. Then HMAC is defined as

$$\mathrm{HMAC}(K, M) = H(K \oplus \mathsf{opad} \| H(K \oplus \mathsf{ipad} \| M)).$$

Here, and in the following, '$\oplus$' is the exclusive-or (XOR) operator.

1.2.4 Ciphertext Correctness Verification

Public key cryptosystems are expected to satisfy a very strong security requirement, namely they must resist so-called Adaptive Chosen-Ciphertext

attacks. Public key cryptosystems do not, in general, satisfy this requirement *per se*. To achieve this level of security, cryptographic hash functions can be used. For details as to how this is done, we refer to e.g., [124].

1.2.5 Proof of Knowledge

Consider the following scenario. A research team decides to organise an internal football betting competition for the upcoming world cup qualification match between Sweden and Denmark. They decide to play on the final score, and each participant pays 50 DKK for each bet. When the result is known, the full amount placed is shared among the winners. But how do the participants place the bets in such a way that nobody knows the other people's bets before they make their own?

They use hash functions: each participant writes down his bet in a text file, followed by a newline character, followed by a sequence of 20 random printable characters (a so-called "nonce"). He then hashes the text file, and publishes the hash. When every participant has placed his bet, the participants publish the text files containing their bets. If someone suspects foul play, he computes the hash of each text file, and compares with the published hashes. The purpose of the nonce is to make it infeasible to perform an exhaustive search among all possible bets. If the hash function is collision and preimage resistant, then it is impossible to cheat.

The above example is a special case of the more general concept of *proof of knowledge protocols*.

1.2.6 Source of Pseudo-randomness

Cryptographic hash functions may be used to construct pseudo-random number generators. Pseudo-random number generators are algorithms that, given a relatively short input, called a *seed*, are able to produce a sequence of numbers that "look random". A few examples are described in [151]. Hash functions are also used as a source of randomness within some cryptographic schemes.

For example, in the *Digital Signature Standard* (DSS) [144], pseudo-random numbers are used extensively, e.g., to compute large primes and to compute a so-called *per message secret number*. The hash function SHA-1 [146] is recommended for this purpose.

As an example, two large primes p and q are needed in the generation of the public/private key pair. The prime q is computed by means of the following algorithm.

1: **repeat**

2: Choose an arbitrary seed x of at least 160 bits (let $|x| = d$).
3: Compute $U \leftarrow H(x) \oplus H((x+1) \bmod 2^d)$, where H is the SHA-1 hash function.
4: Compute $q \leftarrow U \vee 2^{159} \vee 1$.
{Now $2^{159} < q < 2^{160}$, since SHA-1 is a 160-bit hash function, and both the 159th and the least significant bits of q are set}
5: **until** q is prime

The second prime, p, is computed as a function of q, the seed, and other values. SHA-1 is also used in the computation of p.

1.2.7 Key Derivation

Key derivation functions (KDFs) are used to construct a number of cryptographically strong keys from an initial bit string, which is expected to contain some randomness, but not necessarily to be suitable as a cryptographic key by itself. In other words, the key derivation function is expected to evenly distribute the entropy of the input string over the output, which will often be of a different length than the input.

Hash functions are often used to construct key derivation functions, e.g., [1, 57, 63, 89, 113, 149, 182]. It can be argued that some of these constructions are bad practice [1, 113]. However, hash functions do seem to be the natural primitive to use in the construction of a KDF.

1.3 Brief history

The first reference to an algorithm which today would be called a cryptographic hash function seems to have been by Wilkes [219], 1968, who mentions a device constructed by Needham. It is not publicly known how Needham's device worked. Purdy was one of the first to describe [165] a practical design; it was based on polynomials over finite fields. Evans, Kantrowitz and Weiss described [66] a method based on a block cipher in the same journal issue. These "hash functions" in fact did not compress; they were intended to be used in a password protection scheme.

Diffie and Hellman invented [58] public key cryptography (Merkle [132], apparently independently, made the same discovery) in 1976. Public key cryptography provided a method for obtaining digital signatures; Rabin [168] proposed to apply a hash function to the message before signing, for increased performance *and* security.

Matyas, Meyer, and Oseas described [127] some block cipher based hash function constructions, which are still in use today. The MDC-2 construc-

tion [135], which produces a $2n$-bit hash function from an n-bit block cipher, was developed at IBM in 1987. Rivest developed the (apparently) first publicly known dedicated cryptographic hash function, MD2 [91, 120], in 1988. In 1989, Damgård [48] and Merkle [134] independently described construction methods for hash functions. MD2 was followed by another design by Rivest called MD4 [170–172] in 1990. MD4 employed the principles proposed by Damgård and Merkle. It was superseded by MD5 [173] in 1991. In 1993, the U.S. National Institute of Standards and Technology (NIST) developed the Secure Hash Standard (SHA) [142] on the basis of MD4 and MD5. Two years later, NIST revised the hash standard [143] with a small change to the algorithm. The new algorithm was called SHA-1, and the first version is often (but not officially) named SHA-0. MD5 and SHA-1 were very popular hash functions, and they are still in widespread use.

Weaknesses in MD4 were published in 1991 [54]. MD5 was partly cryptanalysed in 1993 [55]. The first collision in MD4 was found by Dobbertin in 1996 [59, 61]. SHA-0 was cryptanalysed in 1998 [31]. However, the runtime of the algorithm was too high to be carried out in practice. In 2004, near-collisions were found for SHA-0 [16]. 2005 proved to be a grand year for hash function cryptanalysis. Improved collision attacks on MD4 and SHA-0 were published [17, 214], and the first collision attack on the full MD5 hash function was described [216]. A practical collision attack on SHA-0 was published later the same year [217], and so was the first collision attack on SHA-1 [215]. The Chinese cryptographer Xiaoyun Wang was a dominant character in most of the attacks that were published in 2005.

All these attacks, and in particular the attacks on MD5 and SHA-1, were quite a shock to the cryptographic community. Although cryptanalysis of hash functions had improved over the previous years, practical or almost practical attacks on the most commonly used hash functions were not expected to appear so soon. NIST had already updated its suite of secure hash functions with the so-called SHA-2 family [146], but these hash functions were developed on the same principles as MD4, MD5, and SHA-1, and therefore the confidence in the SHA-2 family was also damaged by the attacks that appeared in 2005.

In response, NIST hosted a Cryptographic Hash Workshop in October 2005, and another one in August 2006. It was decided to initiate a public competition to develop a new set of hash functions, to be named SHA-3, to augment the existing Secure Hash Standard [146]. The call for candidate algorithms was made in November 2007 [141]. The deadline for candidate submissions was October 31, 2008.

Chapter 2

The Merkle-Damgård construction

The majority of hash functions proposed in the past have been based on the so-called *Merkle-Damgård construction*. In this chapter, we describe the Merkle-Damgård construction and discuss its properties. In the past few years, a number of attacks and weaknesses of the Merkle-Damgård construction have been described. We present some of these as well.

2.1 Introduction

As we have seen, a hash function in principle accepts messages of arbitrary length, and produces a fixed-length output. How does one obtain compression from strings of any length, to strings of a fixed, short length?

The most common method used in practice is to iterate over a so-called *compression function*, that takes a fixed-length input and produces an output of a fixed, but shorter, length. The output size is n bits. If the input size is $b > n$, then we may think of the compression function as a function f with the interface $\{0,1\}^n \times \{0,1\}^{b-n} \rightarrow \{0,1\}^n$, hence, as a function accepting two inputs, one of size n bits, and one of size $b - n$ bits. In the following we let $\mu = b - n$. To hash a message M of size N bits, we may split M into a number of chunks, or *message blocks*, of size μ bits each, and then process each block one at a time using f. The first input to f will then be a state, that is "updated" by the second input, the message block. The output is the new, updated, state.

To be more precise, a hash function H may iterate a compression function f as follows. Let M be the message to be hashed, and assume that it is split into t message blocks $m_1, \ldots, m_t$. Define an *initial state* (or *initial value*) iv

which is fixed for the hash function H. Let $h_0 = \text{iv}$. Then, for each i from 1 to t do the following:

$$h_i \leftarrow f(h_{i-1}, m_i). \tag{2.1}$$

The last state, h_t, is then the output of the hash function, i.e., $H(M) = h_t$. We call the values h_i *chaining values*, *chaining variables*, or *intermediate hash values*, depending on the context.

But what if N, the size of the message M, is not a multiple of μ? Then we need a second function, which we shall call pad_μ, that enlarges M so that its length is a multiple of μ. Hence, instead of feeding M to H directly, we feed $\text{pad}_\mu(M)$ to H. This means, of course, that pad_μ must not produce collisions, because if it did, then it would mean a collision for H as well.

This also means that *every* message must be padded, even if its length is already a multiple of μ. If it was not, then it would be easy to find a collision: Choose an arbitrary message M of length not a multiple of μ. Pad M to M^*, i.e., $M^* = \text{pad}_\mu(M)$. Since M^*, considered as a message in its own right, would not be padded, M and M^* will collide.

In the following, we think of pad_μ as being part of H, meaning that H takes care of calling pad_μ. A possible definition of pad_μ is the following.

Definition 2.1. The padding function pad_μ padding messages to a multiple of μ bits is defined as follows: given input M of length N bits, append the bit '1' to M, and then append $(-N-1) \bmod \mu$ '0' bits. This clearly ensures that the length of the padded version of M is a multiple of μ, and also that pad_μ does not produce collisions.

To see why no collisions are produced by this definition of pad_μ, note that given any output of pad_μ, we can uniquely restore the input by removing all trailing '0' bits as well as the last '1' bit. If we did not always include the '1' bit in the padding, then we would not know how many of the trailing '0' bits came from the padding, and how many were part of the original message.

We are getting close to a description of the Merkle-Damgård construction, but we are not quite there yet. The strength of the Merkle-Damgård construction lies in the fact that there is a reduction proof, that reduces the collision resistance of the *hash function* to the collision resistance of the *compression function*. This security proof requires a slightly more complicated padding function. Consider the construction (2.1), with the padding function of Definition 2.1. A collision may be obtained by the use of a so-called *fixed point*, i.e., an input pair (h, m) to f such that $f(h, m) = h$. If such a pair is found, and $h = \text{iv}$, then the two messages m and $m\|m$ collide in H, but no collision for f has been found.

To overcome this problem, the padding function is changed so that it includes the length of its input in the padding. We give the padding function

an additional subscript τ, whose role is explained later (recall that μ is the message block length).

Definition 2.2. The padding function $\text{pad}^*_{\mu,\tau}$ is defined as follows: given input M of length N bits, append the bit '1' to M, and then append $(-N - \tau - 1) \bmod \mu$ '0' bits. Finally, append a τ-bit representation of N. This clearly ensures that the length of the padded version of M is a multiple of μ, and also that $\text{pad}_{\mu,\tau}$ does not produce collisions.

Since it must be possible to encode N using τ bits, τ in fact defines the upper limit $2^\tau - 1$ to the length N of inputs to the padding function. Although a hash function is expected to accept messages of arbitrary length, in practice it is usually fine if there is a very high upper limit on the message length. For example, most hash functions in common use have $\tau \geq 64$.

We note that when the length of the input is included in the padding, we no longer need to append a '1' bit before appending '0' bits; the input can be reconstructed simply as the first N bits of the output, where N is obtained from the last τ bits of the output. However, it is customary to include the '1' bit in the padding anyway.

The practice of including the length of the original message in the padding is often called *MD-strengthening* (due to Lai and Massey [115]).

With the padding function of Definition 2.2, the construction described above is identical to our definition of the Merkle-Damgård construction. For completeness, we define the Merkle-Damgård construction from scratch (see also Figure 2.1).

Construction 2.1 (The Merkle-Damgård construction)**.** The Merkle-Damgård construction defines a hash function H given some value of τ, an initial value iv, and a compression function $f : \{0,1\}^n \times \{0,1\}^\mu \to \{0,1\}^n$, as follows. Let the input be M, and let $M^* = \text{pad}^*_{\mu,\tau}(M)$, where $\text{pad}^*_{\mu,\tau}$ is as defined in Definition 2.2. Split M^* into blocks $m_1, \ldots, m_t$, let $h_0 = \text{iv}$, and compute $h_1, \ldots, h_t$ sequentially as

$$h_i \leftarrow f(h_{i-1}, m_i) \quad \text{for } 1 \leq i \leq t.$$

Finally, let $H(M) = h_t$.

It is easy to prove that if one has found a collision for H, then one has found a collision for f. In other words, if f is collision resistant, then H is as well. Due to the existence of this security proof, as well as its relative simplicity, the Merkle-Damgård construction has for many years been a very popular construction method for hash functions.

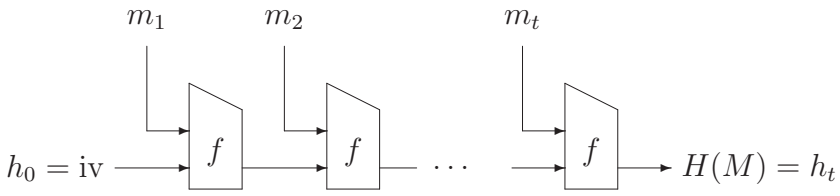


Figure 2.1: The Merkle-Damgård construction.

Later, it will be convenient to have a short notation for the action of "partially" hashing a message. Let M be a message whose length is a multiple of μ bits, and let h be an n-bit chaining value. Then we shall write $F(h, M)$ as a shorthand for $f(f(\cdots f(h, m_1)\cdots), m_t)$, where $M = m_1 \| \cdots \| m_t$, and each m_i is μ bits in length.

The Merkle-Damgård construction is called the Merkle-Damgård construction because R. Merkle [134] and I. Damgård [48], apparently independently, came up with constructions having the mentioned type of security proof. The construction as defined here more closely resembles Merkle's proposal than Damgård's, but since they are fairly similar, and Damgård was the first to give a formal definition of collision resistance, the construction is usually attributed to both.

2.2 Attacks and weaknesses

The Merkle-Damgård construction provably extends collision resistance of the compression function f to collision resistance of the hash function H. However, some other properties are not inherited by H from f. Moreover, some attacks and weaknesses of the Merkle-Damgård construction have been identified. This does not mean that the Merkle-Damgård construction is "broken", however, since the purpose of the construction from the very beginning was only to prove collision resistance. Still, the attacks and weaknesses are seen by some as an indication that the Merkle-Damgård construction does not have all the desired properties of a hash function construction.

In this section, we describe some of the known attacks and weaknesses of the Merkle-Damgård construction. We introduce the term *generic attacks* for attacks on a construction method, where the underlying primitive (e.g., compression function) is assumed to be secure in all respects. An attack that exploits weaknesses in an underlying primitive is called a *shortcut attack*.

2.2.1 Property preservation

The term *property preservation* refers to the ability of a hash function construction to extend properties of the compression function to the full hash function. We have seen that the Merkle-Damgård construction preserves collision resistance. What about other properties?

Assume f is computationally indistinguishable from a random oracle truncated to n bits. We call such a function a *pseudorandom oracle*. This f is a secure compression function. No attacks better than birthday and brute force attacks exist. In a sense, f is *ideal*. Does the Merkle-Damgård construction preserve this "ideality"? Unfortunately, it does not.

A simple counter-example is the following [37]. Assume that f is a pseudorandom oracle, and that we can call f and the Merkle-Damgård hash function H based on f. We want to do it in such a way as to prove that H is not random. First, choose a one-block message m_1, and define $m_1^* = \text{pad}_{\mu,\tau}^*(m_1)$. Let $y = H(m_1)$. Then, let $z = f(y, m_2)$ for a message block m_2 such that $m_1^* \| m_2 = \text{pad}_{\mu,\tau}^*(M)$ for some message M. Now, we know that $H(M) = z$ without ever asking this query. Hence, the outputs of H are not independent, and therefore H cannot be pseudorandom. This observation is related to two weaknesses described in the following subsection.

We shall see below (Section 2.2.4) that the Merkle-Damgård construction also does not preserve second preimage resistance. A much simpler counterexample [5], however, proves that the Merkle-Damgård construction does not preserve second preimage resistance *nor* preimage resistance: assume f is a pseudorandom oracle, with the exception that $f(\text{iv}, m) = \text{iv}$ for all m. In the definition of preimage and second preimage resistance, it is assumed that the attacker is given a randomly chosen image or first preimage. Therefore, this f is still preimage and second preimage resistant (in effect, the resistance degrades by a factor about $1/(1 - 2^{-n})$). However, $H(M) = \text{iv}$ for all M, and therefore preimages and second preimages of the *hash function* can be found in constant time.

Much research has in recent years been devoted to designing constructions that preserve as many properties of the compression function as possible. Some examples are [4, 5, 8, 32, 37]. In Section 3.3.4 we describe some of these.

2.2.2 Length extension

If two messages M and M^* of the same length collide in H, and H is a Merkle-Damgård-based hash function, then it is possible to construct many suffices y such that $M \| y$ and $M^* \| y$ also collide. The suffix y is under almost

complete control of the attacker – the only restriction is that the first bits must be identical to the padding, so that $\text{pad}_{\mu,\tau}(M)$ is a prefix of $M\|y$, and likewise for M^*. Hence, a single collision provides the ability to produce an almost arbitrary number of collisions. This is often seen as a weakness of the Merkle-Damgård construction.

Another related weakness is the fact that given $H(M)$ and $|M|$, but not M itself, one is able to compute $H(M\|y)$ for any y with the same property as above.

It is not clear who first described these weaknesses. They are not mentioned in the Crypto '89 papers [48, 134] by Damgård and Merkle. In the Handbook of Applied Cryptography [131, Example 9.64], the second weakness above is indirectly described. Both weaknesses have been folklore knowledge for many years.

2.2.3 Multi-collisions

The Merkle-Damgård construction allows so-called multi-collisions to be found more easily than one would expect for an ideal hash function. Multi-collisions are now defined.

Definition 2.3 (Multi-collision)**.** Given the hash function H, an r-collision is a set $\{m_1, \ldots, m_r\}$ of r messages that all hash to the same value, i.e., $H(m_1) = \ldots = H(m_r)$. A 2-collision is what is usually merely called a collision.

The expected complexity of finding an r-collision in an ideal hash function is often stated as $2^{(r-1)n/r}$, but this expression is simplified. A more precise estimate is $\left(r!^{1/r}\right) 2^{(r-1)n/r}$ [56, 202].

At Crypto 2004, Joux described a method for constructing multi-collisions in hash functions based on the Merkle-Damgård construction [90]. We note that already in 1985, Coppersmith [35] used the same method to efficiently forge a digital signature scheme by Davies and Price [50]. However, Coppersmith did not generalise the idea.

The method allows one to find a 2^t-collision in time $t2^{n/2}$, namely by finding t regular (2-)collisions. We describe the attack here, and we note that the attack applies to many other conceivable types of iterated hash functions, not just to Merkle-Damgård.

Let $\mathcal{C}$ be an algorithm that (using the birthday attack or some other method) finds collisions in f given some chaining input. Let the initial value of H be h_0. Obtain a collision (m_1, m_1^*) from $\mathcal{C}$ with h_0 as chaining input, i.e., $h_1 = f(h_0, m_1) = f(h_0, m_1^*)$. Now, obtain a second collision (m_2, m_2^*) from $\mathcal{C}$ with h_1 chaining input. Repeat this t times. We now have t pairs of

messages, $(m_1, m_1^*), (m_2, m_2^*), \ldots, (m_t, m_t^*)$, such that we can form a t-block message consisting of a member from each pair, and irrespective of how we make the choice of members, we get the same hash value under H. See Figure 2.2. There are 2^t ways to form such a message, an hence we have a

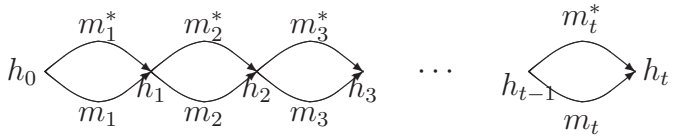


Figure 2.2: Joux's multi-collision attack on the Merkle-Damgård construction.

2^t-collision. The complexity of the attack is t times the complexity of finding a single collision with $\mathcal{C}$, which is at most $2^{n/2}$ for any hash function. Hence, the complexity is $t2^{n/2}$. This is much less than the ideal complexity, which approaches 2^n quickly as t increases.

2.2.4 Second preimage attack

At Eurocrypt 2005, Kelsey and Schneier presented [94] a second preimage attack on all hash functions based on the Merkle-Damgård construction. The complexity of the attack decreases with the size of the given (first) preimage.

Expandable messages

The attack uses so-called *expandable messages*. An expandable message is in fact a set B of messages $\{M_1, M_2, \ldots, M_k\}$ of different lengths, such that given some initial value h, we have $F(h, M_i) = F(h, M_j)$ for any (i, j), $1 \leq i, j \leq k$. Therefore, in a slight abuse of notation, we shall write $F(h, B)$ in short. If B contains messages of any length from a to b blocks (both inclusive), then we say that we have an (a, b)-expandable message. Kelsey and Schneier give two methods of constructing expandable messages that both have complexity around $2^{n/2}$.

The first method is based on fixed points for the hash function (we note that Dean already in 1999 described [53] a related but slightly less practical attack). A fixed point is a pair (h, m) such that $f(h, m) = h$. Fixed points can be found very efficiently (corresponding to one call of f) when the compression function is based on the Davies-Meyer mode (Construction 3.1, see Section 3.1). Many compression functions, including those of the MD4 family, are. To construct an expandable message based on fixed points, one does the following. Find $2^{n/2}$ fixed points (h_i, m_i). Note that the values h_i are generally not under the control of the attacker. Now, given an initial value

iv, compute about $2^{n/2}$ intermediate hash values $f(\text{iv}, m'_i)$ using arbitrary message blocks m'_i. With good probability, there is a match between the chaining values produced by the message blocks m'_i and the chaining values that form fixed points. This means that some message pair (m'_i, m_j) exists such that $f(\text{iv}, m'_i) = F(\text{iv}, m'_i \| m_j) = F(\text{iv}, m'_i \| m_j \| m_j) = \ldots$. Now we have a $(1, \infty)$-expandable message, constructed in time about $2^{n/2+1}$. (See also the description of a meet-in-the-middle attack in Section 4.1.4.)

If fixed points cannot be efficiently found, then a variant of Joux's multi-collision attack, as described in the previous section, can be applied. Assume we have an algorithm $\mathcal{C}$ that takes as input a positive integer ℓ and a chaining value h, and returns a collision (M_0, M_1) such that $F(h, M_0) = F(h, M_1)$, and M_0 is a one-block message and M_1 is an ℓ-block message. We call this a $(1, \ell)$-collision. On a hash function where $\mu \geq n/2$, collisions of this kind are no harder to find than, say, collisions of two one-block messages.
We construct a $(k, 2^k + k - 1)$-expandable message starting from h_0 as follows. For each i from 0 to $k-1$, find a $(1, 2^i + 1)$-collision starting from h_i, where the common output of F in the ith step is h_{i+1}. Let the ith collision be (M_0^i, M_1^i), $0 \leq i < k$. Now, $F(h_0, M_{b_0}^0 \| \cdots \| M_{b_{k-1}}^{k-1}) = h_k$ for any bit-vector $b = (b_0, \ldots, b_{k-1})$. The time required to construct the expandable message is around $k2^{n/2}$. To form a message of length t blocks (where $k \leq t \leq 2^k + k - 1$), view $\tilde{t} = t - k$ as a k-bit binary number, and let b_i be the ith bit of $\tilde{t}$, where counting starts from the least significant bit. The length of the resulting message is $\sum_{i=0}^{k-1}(1 + b_i 2^i) = k + \sum_{i=0}^{k-1} b_i 2^i = k + \tilde{t} = t$.
An example is in order.

Example 2.1. Expandable message
To construct a $(4, 2^4 + 4 - 1)$-expandable message B (i.e., $k = 4$) starting from h_0, do the following.

1: Find a $(1, 2)$-collision (M_0^0, M_1^0) starting from h_0, with $h_1 = F(h_0, M_0^0) = F(h_0, M_1^0)$.
2: Find a $(1, 3)$-collision (M_0^1, M_1^1) starting from h_1, with $h_2 = F(h_1, M_0^1) = F(h_1, M_1^1)$.
3: Find a $(1, 5)$-collision (M_0^2, M_1^2) starting from h_2, with $h_3 = F(h_2, M_0^2) = F(h_2, M_1^2)$.
4: Find a $(1, 9)$-collision (M_0^3, M_1^3) starting from h_3, with $h_4 = F(h_3, M_0^3) = F(h_3, M_1^3)$.

Now $F(h_0, M_{b_0}^0 \| M_{b_1}^1 \| M_{b_2}^2 \| M_{b_3}^3) = h_4$ for all bit-vectors $b = (b_0, b_1, b_2, b_3)$.
A message of length $t = 15$ blocks may be formed as follows. Let $\tilde{t} = t - k = 11$, which is `1011` in binary. Let $b = (1, 1, 0, 1)$ (note that the order of the bits is reversed). Hence, the 15-block message from B is $M_1^0 \| M_1^1 \| M_0^2 \| M_1^3$. To check, note that M_1^0 has length 2 blocks, M_1^1 has length 3 blocks, M_0^2 has length 1 block, and M_1^3 has length 9 blocks. In total, 15 blocks.

Finding the second preimage

Recall that we are describing a second preimage attack. Say the first preimage $M = m_1 \| \cdots \| m_t$ produces the intermediate hash values $w_1, \ldots, w_t$, where $w_t = F(\text{iv}, M)$. Here, we assume for simplicity that the length of M is a multiple of μ bits. If this is not the case, then we may ignore the message block m_t for a while and use $t - 1$ instead of t below.

First, we find an (a, b)-expandable message B (with b being less than, but approximately equal to t) starting from $h_0 = \text{iv}$, and with $v = F(h_0, B)$. Then, to find a second preimage, we search for a *linking message block* d such that $f(v, d) = w_i$ for some i, $a < i \leq b + 1$. The second preimage will have the form $M^* = B \| d \| m_{i+1} \| \cdots \| m_t$. To ensure that padding is the same for M and M^*, the length of M^* has to be the same as the length of M, so we have to expand B to a length of $i - 1$ blocks. Since $F(\text{iv}, B \| d) = F(\text{iv}, m_1 \| \cdots \| m_i)$, the two messages produce the same intermediate hash values from the ith value on, and since $|M| = |M^*|$, padding is also the same for the two messages. Therefore they collide, and M^* is a second preimage of M.

Complexity

As mentioned, the complexity of finding the expandable message depends on the method; if fixed points are used, then the complexity is about $2^{n/2+1}$. The complexity of the second method using a multi-collision, in turn, depends on the size of the expandable message. In order to match the size of the first preimage, one would construct a $(k, 2^k + k - 1)$-expandable message, with $2^k \approx t$. This takes time approximately $k2^{n/2}$.

With $2^k \approx t$, finding the linking block d takes expected time close to 2^{n-k}, because there are about 2^k possible intermediate values that may be hit, and the probability of hitting each of them is about 2^{-n}. Hence, it is clear that the attack gets more efficient as t increases. However, computing the intermediate hash values w_i is a task whose cost is placed on the attacker, since in a second preimage attack, we assume the attacker is given only the first preimage and its final hash. Computing the t intermediate hash values takes time about t.

In total, the complexity of the attack when fixed points are used to find the expandable message is about $2^{n/2+1} + 2^{n-k} + 2^k$, again with $2^k \approx t$. If the expandable message is constructed using a multi-collision, then the total complexity of the attack is about $k2^{n/2} + 2^{n-k} + 2^k$. See Table 2.1. We see that approximately $2^{n/2}$ is a lower bound on the attack, regardless of the length of the first preimage. Also, the length of the first preimage is upper

Table 2.1: Complexities of the second preimage attack assuming a first preimage of length about 2^k message blocks, using two different methods of constructing the expandable message.

Expandable message method	Complexity
Fixed points	$2^{n/2+1} + 2^{n-k} + 2^k$
Multi-collision	$k2^{n/2} + 2^{n-k} + 2^k$

bounded by the value of τ in the padding function of Definition 2.2.

The memory requirements are up to about $2^{n/2}$ for finding the expandable message using fixed points, but may be negligible (for the expandable message construction) if the method using a multi-collision is used. The attacker also needs to store the t intermediate hash values of the first preimage.

As an example, in SHA-256 the maximum message length is about 2^{55} blocks. Fixed points can be efficiently found in SHA-256. Hence, if one ever comes across a first preimage of length about 2^{55} message blocks, a second preimage can be found in time about $2^{129} + 2^{256-55} + 2^{55} \approx 2^{201}$. A brute force second preimage attack takes time 2^{256}.

2.2.5 The "Nostradamus" attack

The Merkle-Damgård construction also allows a non-standard type of attack sometimes referred to as the Nostradamus attack, or less exotically as a chosen target forced prefix attack (CTFP). The particular technique described in this section to carry out the Nostradamus attack is called the "herding attack" [93].

In a Nostradamus attack, the attacker claims to be able to predict future events. He does this by publishing a hash value T before the event takes place. After the event, he publishes a message M, containing enough information to "prove" that the attacker knew about the event before it took place, and such that $H(M) = T$.

For an ideal hash function, this "attack" has expected complexity 2^n – assuming that the attacker does not possess the Nostradamus-like ability to predict future events. The attack resembles a preimage attack. However, in this case, the attacker is allowed to choose the target image, but must produce a message whose contents is not under his complete control.

On Merkle-Damgård hash functions, the attack can be carried out as follows. First, there is a *precomputation phase*, which produces the hash value T to be published. After the event has taken place, the *online phase* can begin.

Precomputation phase

The attacker finds collisions forming a binary tree with nodes labelled with intermediate hash values, and edges labelled with message block values. The tree has a structure such that every pair of siblings in the binary tree collides (under f) to the intermediate hash value nearer to the root. The collision is of the form $(h, m), (h^*, m^*)$, where $h \neq h^*$. At the root is the intermediate hash value T^*. With d being a padding block containing correct padding (hence, the length of the final message must be known at this point), we have $f(T^*, d) = T$, where T is the hash value that is published.

The depth of the tree is k, so there are 2^k leaves, all having different labels. From each leaf, there is a path of message blocks leading to the root. A final message length greater than k blocks must be chosen, and the padding block d will contain correct padding for a message of this length. See Figure 2.3.

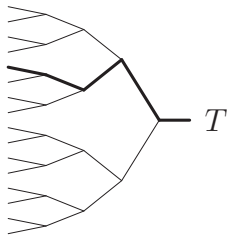


Figure 2.3: The tree with a path highlighted.

The tree can be constructed in time about $2^{n/2+k/2}$ as follows. For each of the 2^k starting values (values of the leaves) h^i, compute $y_{i,j} = f(h^i, m_j)$ for $2^{n/2-k/2}$ arbitrary message blocks m_j. Now, for each pair of leaves $(h^i, h^{i'})$, the probability that there is a message pair $(m_j, m_{j'})$ such that $y_{i,j} = y_{i',j'}$, is about $(2^{n/2-k/2})^2 \cdot 2^{-n} = 2^{-k}$. Hence, we expect that each starting value collides with one other starting value (on average), forming a pair of siblings. These figures are rough, but close enough.

The next level in the tree can be produced twice as fast, etc., so the total complexity is in fact about $2^{n/2+k/2+1}$. The leaves can be stored in a list L, thus containing 2^k intermediate hash values. Having constructed the tree, the hash value T can be published, and the precomputation phase is completed.

Online phase

When the event has taken place, a (sub-)message M^* is produced containing information "proving" prior knowledge of the event. Compute $v = F(\text{iv}, M^*)$

(we assume for simplicity that the length of M^* is a multiple of μ). Now, search for a (random) linking block m_{L} such that $f(v, m_{\mathrm{L}}) \in L$. This is expected to require 2^{n-k} calls to f. From the intermediate hash value in L there is, as mentioned, a path that leads to the root of the tree constructed in the precomputation phase. If this path is called M_{P}, then we have that $H(M^* \| m_{\mathrm{L}} \| M_{\mathrm{P}}) = T$ (d contains all padding). See Figure 2.4.

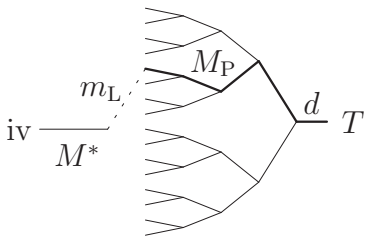


Figure 2.4: The Nostradamus attack.

Complexity

The complexity of the precomputation phase is about $2^{n/2+k/2+1}$. The complexity of the online phase is about 2^{n-k}. Balancing the two phases yields $k \approx n/3$, resulting in a total complexity of $2^{2n/3+1}$ for the precomputation phase, and $2^{2n/3}$ for the online phase. This is below the expected complexity for an ideal hash function of 2^n.

Variants

In some cases, other (more efficient) methods of carrying out the Nostradamus attack can be applied [93]. For instance, one can use the multi-collision and second preimage attacks described above. The multi-collision attack can be used in scenarios where the message that is published in the end may consist of a sequence of blocks containing information such as true/false, yes/no, etc. Combined with a document scripting language such as PostScript, it can also be used to predict, e.g., stock prices. See also Section 4.1.2.

The second preimage attack can be used instead of the herding attack described above, and it gives the attacker complete freedom in choosing a large part of the message, but the final message will become very long. The attacker simply chooses a very long first preimage, hashes it, and publishes the hash value. In the online phase, he constructs the desired message prefix, partially hashes it to obtain the intermediate hash value y, and carries out the second preimage attack, assuming y is the initial value. The complexity of

the attack may be lower than that of the herding attack, if the hash function accepts very long messages. The final message will contain many message blocks that are equal to those in the first preimage. Thus, the attacker has a large amount of freedom which he may be able to apply in producing a convincing message, despite its being very long.

Chapter 3

Hash function design

In the beginning of hash function history, hash functions were often based on a cryptographic primitive called a *block cipher*. A block cipher is a combined encryption/decryption primitive. In encryption mode, the block cipher accepts as input a key K and a plaintext block m, and outputs a ciphertext block $c = E(K, m)$, often written $c = E_K(m)$. The function E is invertible when K is known, and hence the ciphertext can be decrypted such that $m = E_K^{-1}(c)$. Without knowing the key K, this task is supposed to be difficult. It is also supposed to be difficult to find K from a number of pairs (m, c), even if m or c may be chosen by an attacker in all pairs. We note that it is always the case that $|m| = |c|$, e.g., $|m| = n$, but furthermore we often assume that $|K| = n$ (or $|K| = 2n$). Hence, a block cipher (in encryption mode) in a sense compresses the $2n$ (or $3n$) bits constituting the key and the plaintext block to n bits of ciphertext.

Most hash functions can be described as being based on a block cipher, but in many cases the block cipher was designed specifically for use in a hash function, and is not necessarily secure in the above sense. We call such hash functions *dedicated hash functions*. This class of hash functions generally includes all hash functions that are not based on some existing primitive, but were rather designed "from scratch". By *block cipher-based hash functions* we mean hash functions based on an existing block cipher, that was designed for encryption/decryption.

Hash functions may also be based on "non-compressing" primitives, for instance permutations. Since a hash function compresses, it may be clear that constructing a hash function from a non-compressing primitive poses an additional challenge compared to the above case.

In this chapter, we discuss all the above mentioned types of hash functions, and we present some specific examples. Three of these were co-designed by the author. We also discuss some alternatives to the Merkle-Damgård

construction.

3.1 Hash functions based on block ciphers

Most hash functions in use today are based on a block cipher, but they are often not considered as such, since the block cipher was designed for the hash function specifically, and was never intended to be used for encryption. This section deals with hash functions whose compression function makes intensive use of an existing block cipher. Typically, the compression function is iterated in Merkle-Damgård mode. Security can in many cases be proven in the so-called *ideal cipher model*, where the underlying block cipher is modelled as an ideal cipher, i.e., a family of random permutations. It is important to note that the ideal cipher model has some limitations. For instance, it is possible to design a hash function construction that is provably collision resistant in the ideal cipher model, but which completely fails to protect against collisions once the block cipher is instantiated [21]. Although this construction is extremely artificial, it gives rise to some doubts about security proofs in the ideal cipher model. However, proving security in this model seems to be the best we can do.

For block cipher-based hash function constructions, the *rate* is defined to be the number of message blocks that are processed per block cipher call, where a message block is assumed to have the same size as the block cipher. More precisely, if n is the size of the block cipher, μ is the size of a message block, and the construction requires t calls of the encryption function per message block, then the rate is $\frac{\mu}{tn}$.

When a block cipher is used for encryption, the key is (usually) first expanded to a number of round keys in what is called the *key schedule*. Once the key schedule has been applied, the encryption can start. Hence, the key schedule is applied at most once for each plaintext that must be encrypted. When the block cipher is used in a hash function, however, the key schedule often has to be applied once for each *message block*. For many block ciphers, applying the key schedule is a fairly time-consuming operation, often requiring about the same amount of time as (or more than) encrypting a plaintext block. Therefore, block cipher-based hash function constructions of rate 1 do not in practice provide hash functions with the same speed as the underlying block cipher. This may be the single most important reason why most hash functions used in practice were designed from scratch, and not based on an existing block cipher. Another important reason is that block ciphers may be covered by export restrictions, whereas dedicated hash functions usually are not. However, the study of constructions turning a block

cipher into a secure hash function remains an important and interesting one.

The simplest type of constructions are the so-called single length constructions, for which the output size of the hash function equals the size of the block cipher. Many (provably) secure constructions of this kind are known [24, 163].

However, single length constructions may not be the most interesting constructions in practice. This is due to the fact that relevant generic attacks on n-bit block ciphers have complexity at least 2^n, whereas, as mentioned, for cryptographic hash functions, the generic birthday collision attack has complexity $2^{n/2}$. Hence, most dedicated block ciphers are designed with n sufficiently large for 2^n to be an acceptable security level, but $2^{n/2}$ may not be. Therefore, in order to construct a cryptographic hash function with sufficiently large output, double length (or larger) constructions are needed. Perhaps not surprisingly, these have turned out to be much more difficult to design than single length constructions.

We start off with a brief discussion of some single length constructions that were shown to be secure in [24, 163], and then we move on to double length constructions. In the following, we let $E_K(m)$ be the function (obtained from some block cipher) that encrypts the plaintext m using the key K.

3.1.1 Single length constructions

As mentioned, many secure single length constructions are known. The most famous and flexible of the secure constructions is (apparently due to [166]) called the Davies-Meyer construction.

Construction 3.1 (Davies-Meyer)**.**

$$f(h, m) = E_m(h) \oplus h.$$

The XOR with h is often called a *feed-forward*. It is unclear when this construction was first described, but it was first proven to be one-way in the ideal cipher model by Winternitz [220], and to be collision resistant (apparently) by Black, Rogaway, and Shrimpton [24]. It has the advantage over most other secure single length constructions that the block cipher key does not need to be of the same size as the plaintext block. For this reason, the construction is often used in dedicated hash function designs, for instance in the SHA-2 family, where the "key" (message block) is twice as large as the "plaintext block" (chaining value). Although it is provably collision and preimage resistant in the ideal cipher model, the construction has a property which may pose a problem in some applications: by setting $h = E_m^{-1}(0)$ (for

any m), one gets a fixed point, i.e., $f(h, m) = h$. We note, however, that h is not under the direct control of the attacker.

Another famous and secure single length construction, apparently first described in [127], is the dual of the Davies-Meyer construction. It is usually named Matyas-Meyer-Oseas (or MMO) after the authors of [127].

Construction 3.2 (Matyas-Meyer-Oseas)**.**

$$f(h, m) = E_h(m) \oplus m.$$

This construction requires that $|h| = |m|$, i.e., that the key size is equal to the block size. (Alternatively, one may apply a function g to the output before using it as a key in the following iteration, such that the size is correct. However, the security of the construction depends on the minimum of the block size and the key size.) Like the Davies-Meyer construction, this construction is collision resistant if the block cipher is secure. Fixed points cannot be easily found. The MMO construction forms the basis of the MDC-2 and MDC-4 double length constructions, which are described in Section 3.1.2.

A third, secure single length construction is the so-called Miyaguchi-Preneel construction, independently proposed by Miyaguchi (et al.) [136] and Preneel [160].

Construction 3.3 (Miyaguchi-Preneel)**.**

$$f(h, m) = E_h(m) \oplus h \oplus m.$$

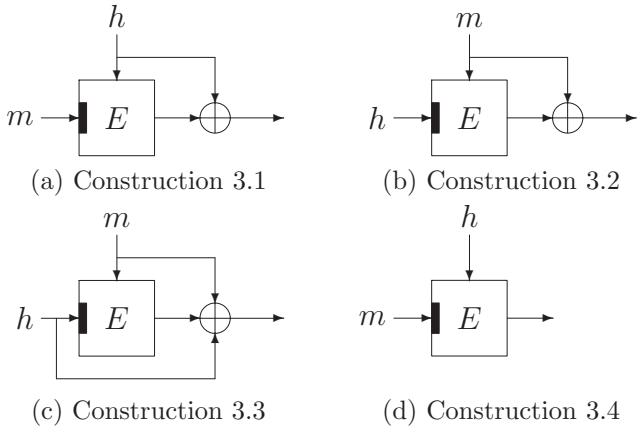
The hash functions N-Hash [136] and Whirlpool [6] are based on the Miyaguchi-Preneel construction.

All these three constructions are provably secure in the ideal cipher model. To be precise, these constructions provide secure compression functions. There are constructions that provide completely *in*secure compression functions, but for which the *hash function* is provably collision resistant. An example is the following construction, which was apparently first proposed by Rabin [167].

Construction 3.4 (Rabin)**.**

$$f(h, m) = E_m(h).$$

As mentioned, a hash function based on Construction 3.4 is provably collision resistant, but preimages and second preimages can be found by a so-called meet-in-the-middle attack (see Section 4.1.4) in time about $2^{n/2}$.



(a) Construction 3.1 (b) Construction 3.2

(c) Construction 3.3 (d) Construction 3.4

Figure 3.1: Some single-length constructions. The "hatch" signifies the key input.

3.1.2 Double length constructions

A double length construction maintains a state of size $2n$ bits, where n is the size of the underlying block cipher. Quite a large number of double length constructions have been proposed in the past, and many of these have also been shown not to be secure. Only recently, double length constructions with provable security started to appear.

A large class of rate 1 double length constructions can be described as follows. View the chaining input as a pair $(h_{i-1}, \tilde{h}_{i-1})$ of n-bit chaining values, and let m_i and $\tilde{m}_i$ be two n-bit message blocks (we assume that the key size is equal to the block size). The two new chaining values, h_i and $\tilde{h}_i$, are computed as

$$\begin{array}{rcl} h_i = f_1(h_{i-1}, \tilde{h}_{i-1}, m_i, \tilde{m}_i) & = & E_A(B) \oplus C \\ \tilde{h}_i = f_2(h_{i-1}, \tilde{h}_{i-1}, m_i, \tilde{m}_i, h_i) & = & E_U(V) \oplus W, \end{array} \tag{3.1}$$

where A, B, C are all linear combinations of the inputs to f_1, and U, V, W are all linear combinations of the inputs to f_2 (note that these include the output of f_1). In [100], all constructions of this type were broken in the sense of both collision and preimage attacks. Specific constructions that can be described as (3.1) include Parallel Davies-Meyer [83], PBGV [161], and the LOKI DBH mode [28].

Hence, it seems that secure double length constructions either have a rate that is less than 1, or the block cipher must accept keys that are larger than the block size. Two examples of the former are MDC-2 and MDC-4 [135], patented by IBM [26] and standardised in ISO/IEC 10118-2 [86]. Let h_{i-1} and $\tilde{h}_{i-1}$ be two n-bit chaining inputs, and let m_i be an n-bit message block. The

MDC-2 construction can then be described as follows (see also Figure 3.2).

Construction 3.5 (MDC-2)**.**

$$\begin{aligned} v &= E_{h_{i-1}}(m_i) \oplus m_i \\ \tilde{v} &= E_{\tilde{h}_{i-1}}(m_i) \oplus m_i, \end{aligned}$$

followed by the computation of the two new chaining variables

$$\begin{aligned} h_i &= v^L \| \tilde{v}^R \\ \tilde{h}_i &= \tilde{v}^L \| v^R, \end{aligned}$$

where v^L and v^R mean the left and right halves of v, respectively.

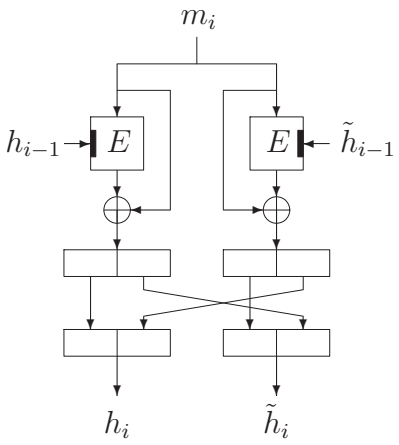


Figure 3.2: The MDC-2 construction.

MDC-2 has rate 1/2. MDC-4 is an up-scaled variant having rate 1/4. We shall not describe MDC-4 here. MDC-2 has been proven [197] collision resistant up to at least $2^{3n/5}$ block cipher calls, where n is the block size of the block cipher. We note that since the double length construction provides $2n$ bits of output, the collision resistance of an ideal hash function of the same size is 2^n. A preimage attack on MDC-2 of complexity about $2^{3n/2}$ was described in [115]. In Section 4.3, we describe a collision attack and improved preimage attacks on MDC-2.

Some examples of constructions, which require that the block cipher accepts keys that are larger than the block size, are given next. Again, h_{i-1} and $\tilde{h}_{i-1}$ are two n-bit chaining inputs, and m_i is a message block.

Construction 3.6 (Tandem Davies-Meyer [115])**.** First, one computes

$$v = E_{h_{i-1} \| m_i}(\tilde{h}_{i-1}).$$

Then, the two new chaining variables are computed as

$$\begin{aligned} h_i &= E_{m_i\|v}(h_{i-1}) \oplus h_{i-1} \\ \tilde{h}_i &= v \oplus \tilde{h}_{i-1}. \end{aligned}$$

Assuming the key size is twice the block size, the rate of this construction is 1/2. No attacks are currently known. A second construction of this type is the following.

Construction 3.7 (Abreast Davies-Meyer [115])**.**

$$\begin{aligned} h_i &= E_{m_i\|\tilde{h}_{i-1}}(\overline{h}_{i-1}) \oplus h_{i-1} \\ \tilde{h}_i &= E_{h_{i-1}\|m_i}(\tilde{h}_{i-1}) \oplus \tilde{h}_{i-1}, \end{aligned}$$

where $\overline{h}$ is the bitwise complement of h.

Abreast Davies-Meyer has also not been broken. A more recent construction of the same type is due to Hirose [82].

Construction 3.8 (Hirose, FSE 2006)**.**

$$\begin{aligned} h_i &= E_{\tilde{h}_{i-1}\|m_i}(h_{i-1}) \oplus h_{i-1} \\ \tilde{h}_i &= E_{\tilde{h}_{i-1}\|m_i}(h_{i-1} \oplus c) \oplus h_{i-1} \oplus c, \end{aligned}$$

where c is a non-zero n-bit constant.

This construction is provably collision resistant in the ideal cipher model. Note that the two block cipher applications use the same key, which means that the key schedule only has to be applied once for each message block. Hirose has proposed other double length constructions with provable security [81].

Knudsen and Preneel suggested to construct multiple length schemes based on error-correcting codes [104–106]. Some instances of these constructions were broken in [218].

3.1.3 A generalisation

Double length constructions (or more generally: multiple length constructions) can be used with not only a block cipher, but with any (secure) fixed input length compression function. An example is to construct a secure 256-bit *external* compression function from a number of secure 128-bit *internal* compression functions.

Peyrin et al. [155] gave lower bounds on the number of internal compression functions needed in a multiple length construction in order to offer optimal collision and preimage resistance for the external compression function. The bounds are attack-based, meaning that the authors give general attacks and state the minimum number of internal compression functions needed for the attacks to fail. We note that the attacks apply to the external compression function, but not necessarily to the hash function.

When considering double length constructions based on internal compression functions mapping from $2n$ to n bits, thus corresponding above to block ciphers with equal key and block size, Peyrin et al. showed that at least five internal compression functions are needed for the external compression function to be secure. With five internal compression functions, however, one *or* two message blocks may be digested per external compression function call, yielding a maximum rate of 2/5. MDC-2, for instance, is not secure in the model used in [155], since a collision in the (external) compression function can be found in time $2^{n/2}$, where 2^n is required for a double length construction to be secure. This is an example of a collision attack on the compression function which does not seem extensible to the hash function, since the attack assumes a large amount of freedom in the choice of chaining inputs.

Peyrin et al. also showed that if the internal compression functions map from $3n$ bits to n bits, then a secure rate 1/2 scheme may exist. We note that Constructions 3.6, 3.7, and 3.8, where the latter is provably collision resistant, are of this kind (assuming n-bit message blocks).

3.2 Permutation-based hash functions

Since hash functions based on block ciphers are somewhat penalised by the frequent need for applying the key schedule, it is tempting to consider hash functions based on a block cipher using only a small, fixed set of keys. Since this means that only a few of the permutations offered by the block cipher will be used, we shall call these hash functions *permutation-based*. The permutations may, of course, also be designed from scratch (see, e.g., Section 5.1). Permutation-based hash functions were first discussed by Preneel, Govaerts, and Vandewalle [162]. We also mention a related class of hash functions, namely those based on random non-compressing functions (i.e., they are not permutations); in some cases, these random functions can be replaced by permutations with a feed-forward of the input.

A large class of permutation-based hash functions was analysed by Black, Cochran, and Shrimpton [23]. This class consists of all the constructions

where a single permutation call is made for each message block (hence, the rate as defined in the previous section is 1), and only a small, fixed set of permutations is used in total. We discuss the results of Black et al. below.

3.2.1 The results of Black, Cochran, and Shrimpton

Let us more precisely define the class of hash functions that Black et al. considered in [23]. Let E_K be an n-bit block cipher using the κ-bit key K, and let $\mathcal{K}$ be a small, fixed-sized, non-empty subset of the 2^κ possible keys. Let the compression function of the hash function be $f : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$, and let $g_1 : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$ and $g_2 : \{0,1\}^n \times \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$ be arbitrary functions. Define f as

$$f(h,m) = g_2(h, m, E_K(g_1(h,m))),$$

where $K \in \mathcal{K}$ may vary for each application of f. In words, g_1 takes the two inputs and produces a single n-bit output, which is fed to the permutation. The output of this permutation, together with h and m, are then fed to g_2, which produces a single n-bit output, which in turn becomes the output of the compression function.

Since g_2 can perform the same computations as g_1, g_2 does not need to be given the output of g_1. This description clearly covers all hash functions in the mentioned class. SMASH [98] is an example of a hash function in this class.

In practice, g_1 and g_2 will be simple and efficiently computable functions, since otherwise we have not gained anything by avoiding invocations of the key schedule of the block cipher. Black et al. described a *query-based* (also called *information theoretic*) collision attack on f in time about linear in n, that works for any choice of functions g_1 and g_2. By a query-based attack, we mean that the complexity of the attack is stated in terms of block cipher calls – hence, calls to g_1 and g_2 are considered to be "free of charge". The attack requires up to about 2^n calls to g_1 and g_2, and it also requires the construction of a tree containing 2^n nodes. Therefore, it may not be possible to carry out the attack in practice, even if its complexity is roughly linear in terms of block cipher calls.

As an obvious next step, we consider permutation-based hash functions of rate less than 1.

3.2.2 The results of Rogaway and Steinberger

Rogaway and Steinberger considered permutation-based hash functions of any rate, also including multiple-length constructions [177]. The model is (in

brief) the following.

The compression function f takes d blocks of n bits each as input, and produces e blocks of n bits as output. It does this by the application of t n-bit permutations $p_1, \ldots, p_t$. The ith permutation p_i is fed with the output of an arbitrary function g_i, which given the dn bits of input to f, and the $(i-1)n$ bits of output from the previous $i-1$ permutation calls, produces a single n-bit output. In the end, an output transformation function G takes all $(d+t)n$ bits of input to f and outputs from all permutation calls, and produces en bits, that become the output of f. See Figure 3.3.

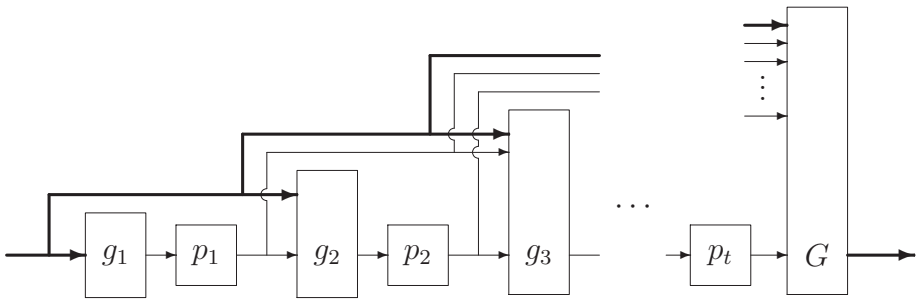


Figure 3.3: The permutation-based compression function f. A thick line means the data path has width dn bits (except the rightmost arrow, which has width en bits), and a thin line means the data path has width n bits.

Rogaway et al. showed [177] that the above construction, which we shall call a (d, e, t)-construction, admits collisions in around $2^{n(1-(d-e/2)/t)}$ queries to one of the permutations p_i. In order for f to compress, we must require that $d > e$. As an example, a $(2, 1, t)$-construction (i.e., a single length construction comparable to those described in Section 3.1.1, but possibly of a lower rate) needs $t \geq 3$ in order to achieve collision resistance at the birthday bound. With $t = 2$, one gets a collision resistance of at most $2^{n/4}$.

With respect to double length constructions ($e = 2$), one can never reach the ideal collision resistance of 2^n (the output size being $2n$ bits) using permutation-based hashing.

Preimages can be found using about $2^{n(1-(d-e)/t)}$ queries. This shows that preimage resistance can never be optimal (since $d > e$). With a $(2, 1, 3)$-construction, one gets an upper bound on the preimage resistance of $2^{2n/3}$. A $(2, 1, 2)$-construction has preimage resistance at most $2^{n/2}$.

Again, we stress that the above bounds are query-based. The true complexity of an attack is likely to be much higher. This (in part) depends on the functions g_i and G. Moreover, Stam [196] discovered a paradox in these complexities: increasing the output size always reduces the collision

and preimage resistance. This clearly cannot be true, since one can always expand the output with a fixed, constant string without affecting collision and preimage resistance. Stam points out that the results of Rogaway and Steinberger rely on the assumption that the outputs of the compression function are uniformly distributed.

3.2.3 Provably secure constructions

Rogaway and Steinberger also (in a different paper [178]) provided a provably secure $(2, 1, 3)$-construction. When based on 128-bit permutations (e.g., permutations obtained from the AES [145]), the construction uses multiplication with the constant $\mathbf{2}$ in the finite field $\mathbb{F}_{2^{128}}$. The compression function performs the following steps (on input h, m):

$$\begin{aligned} y_1 &= p_1(g_1(h, m)) = p_1(h + \mathbf{2} \cdot m) \\ y_2 &= p_2(g_2(h, m, y_1)) = p_2(y_1 + \mathbf{2} \cdot h + \mathbf{2} \cdot m) \\ y_3 &= p_3(g_3(h, m, y_1, y_2)) = p_3(y_2 + \mathbf{2} \cdot h + m) \\ f(h, m) &= G(h, m, y_1, y_2, y_3) = h + y_1 + y_2 + \mathbf{2} \cdot y_3. \end{aligned}$$

All arithmetic takes place in the field $\mathbb{F}_{2^{128}}$.

The above construction has collision resistance almost (but not quite) $2^{n/2}$. Its preimage resistance is also close to, but not quite, $2^{2n/3}$, which, as mentioned, is optimal for a $(2, 1, 3)$-construction. These are proven lower bounds, and the (modest) difference between proven bounds and optimal bounds (according to [177]) may well be an effect of shortcomings in the analysis.

Shrimpton and Stam [195] described a $(2, 1, 3)$-construction assuming random n-bit to n-bit ("length-preserving") functions f_1, f_2, f_3 instead of ideal permutations:

$$\begin{aligned} y_1 &= f_1(g_1(h, m)) = f_1(h) \\ y_2 &= f_2(g_2(h, m, y_1)) = f_2(m) \\ y_3 &= f_3(g_3(h, m, y_1, y_2)) = f_3(y_1 \oplus y_2) \\ f(h, m) &= G(h, m, y_1, y_2, y_3) = y_2 \oplus y_3. \end{aligned}$$

This construction has provable collision resistance around $2^{n/2}/n$. The analysis also holds if the two first random functions are instantiated by ideal permutations with a feed-forward. If all three random functions are replaced with permutations with a feed-forward, then security has not been proved,

but no attack is known. For completeness, this results in the following construction.

$$\begin{aligned} y_1 &= p_1(g_1(h,m)) = p_1(h) \\ y_2 &= p_2(g_2(h,m,y_1)) = p_2(m) \\ y_3 &= p_3(g_3(h,m,y_1,y_2)) = p_3(y_1 \oplus y_2 \oplus h \oplus m) \\ f(h,m) &= G(h,m,y_1,y_2,y_3) = y_1 \oplus y_3 \oplus h. \end{aligned}$$

Stam [196] described a $(2,1,2)$-construction based on random functions instead of permutations, with almost optimal collision resistance of around $2^{n/2}/n$. This significantly breaks the upper bound of $2^{n/4}$ given by Rogaway and Steinberger. The construction is quite unusual; it maintains an n-bit chaining value, but uses $(3n/2)$-bit length-preserving random functions. The chaining value is first expanded to $3n/2$ bits by appending $n/2$ '0' bits, and the output of the final random function is truncated from $3n/2$ bits down to n bits.

One of the random functions can be replaced by an ideal permutation with a feed-forward without affecting the security proof. We describe the construction assuming random functions f_1 and f_2. The chaining value h is of size n bits, and the message block m is of size $3n/2$ bits. h is padded with $n/2$ '0' bits to h^+.

$$\begin{aligned} y_1 &= f_1(g_1(h,m)) = f_1(m) \\ y_2 &= f_2(g_2(h,m,y_1)) = f_2(h^+ \oplus y_1) \oplus h^+ \\ f(h,m) &= \mathsf{trunc}_n(y_2). \end{aligned}$$

By $\mathsf{trunc}_n(y_2)$ we mean y_2 truncated to n bits by removing all but the last n bits. Whether or not security is affected if one keeps the $n/2$ bits that are discarded in this truncation, instead of appending $n/2$ '0' bits in the following application of the compression function, is not clear.

3.3 Alternatives to Merkle-Damgård

Due to the weaknesses found in the Merkle-Damgård construction in recent years, as well as the large number of shortcut attacks that appeared on many widely deployed hash functions, many researchers have considered alternative construction methods to improve security.

In this section we describe some of the proposed alternatives. We start off with a construction method that defines a collision resistant hash function based on weaker assumptions on the underlying compression function than

the assumptions required for the Merkle-Damgård construction. Then, we describe a construction assuming a compression function that produces a larger output than the hash function, thus complicating generic attacks. We also describe the related checksum-based hash functions, and we mention construction methods that aim to preserve other properties of the underlying compression function than merely collision resistance. Finally, we describe a relatively new hash function construction called the sponge.

3.3.1 Knudsen-Thomsen, Secrypt 2006

Knudsen and this author [109, 110] proposed an alternative to the Merkle-Damgård construction, that requires access to the compression function. The goal was to improve resistance to some generic attacks, and also to reduce the amount of freedom that an attacker has in carrying out shortcut attacks on the hash function.

The proposal

The construction proposed by Knudsen and Thomsen can, slightly simplified, be described as follows.

Construction 3.9. A hash function H based on a compression function $f : \{0,1\}^n \times \{0,1\}^\mu \to \{0,1\}^n$ with $\mu > n$ is defined. Let the message to be hashed be M, padded using the function pad_μ (note: no length padding, see Section 2.1) and separated into t message blocks $m_1, \ldots, m_t$, each of $\mu - n$ bits. It is required that $t < \min(2^n, 2^{\mu-n})$. Iterate the compression function f as follows. With h_0 being the initial value of the hash function (e.g., the all-zero string), do for i from 1 to t

$$h_i = f(\langle i \rangle_n, m_i \| h_{i-1}).$$

Here, $\langle i \rangle_n$ means the n-bit representation of i. When h_t has been computed, let $H(M) = f(0, \langle t \rangle_{\mu-n} \| h_t)$.

This construction has the following property.

Theorem 3.1. *If f is resistant to collisions of the form $((x, y), (x, y^*))$, then Construction 3.9 defines a collision resistant hash function.*

Proof. Assume a collision for H, defined as in Construction 3.9, has been found, i.e., a pair (M, M^*) such that $M \neq M^*$ and $H(M) = H(M^*)$. Let t and t^* denote the number of blocks in M and M^*, respectively, and let these blocks be called m_i and m_i^*. h_i and h_i^* are the intermediate hash

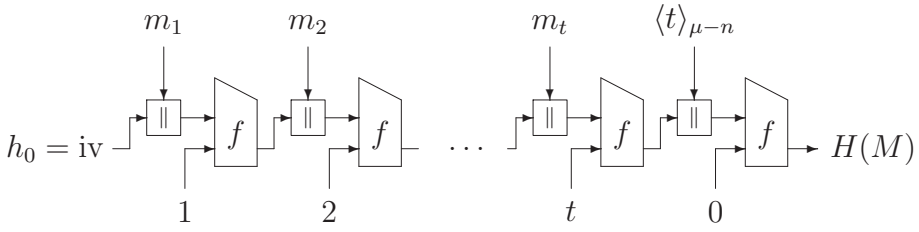


Figure 3.4: The Knudsen-Thomsen construction (Construction 3.9).

values of H when processing M and M^*, respectively. If $t \neq t^*$, then we have $f(0, \langle t \rangle_{\mu-n} \| h_t) = f(0, \langle t^* \rangle_{\mu-n} \| h^*_{t^*})$, and hence the theorem is proved in this case. Assume now that $t = t^*$. If $h_t \neq h^*_t$, then the result follows again. If $h_t = h^*_t$ then it follows that for some j, $0 < j \leq t$, we have $f(\langle j \rangle_n, m_j \| h_{j-1}) = f(\langle j \rangle_n, m^*_j \| h^*_{j-1})$, but $(m_j, h_{j-1}) \neq (m^*_j, h^*_{j-1})$, because M and M^* are different. Thus, the theorem is proved. □

Hence, the collision resistance of Construction 3.9 is guaranteed assuming a weaker condition on the compression function, than what is required in the Merkle-Damgård construction.

For hash functions where the underlying compression function is a block cipher employed in Davies-Meyer mode (Construction 3.1), i.e., $f(h, m) = E_m(h) \oplus h$, Knudsen and Thomsen proposed the following variant. Let

$$h_i = E_{m_i \| h_{i-1}}(\langle i \rangle_n) \oplus h_{i-1},$$

and let $H(m) = E_{\langle t \rangle_{\mu-n} \| h_t}(0) \oplus h_t$. Hence, the input to the block cipher is not fed forward, but instead the chaining input is. The motivation for this change would be that feeding forward a counter does not result in a one-way compression function if the counter is known. On the other hand, for an ideal block cipher it is difficult to find h, m, c given y such that $E_{m \| h}(c) \oplus c = y$. Therefore, we shall assume here that Construction 3.9 is used also when f is a block cipher in Davies-Meyer mode.

Theorem 3.1 means that for H to be collision resistant, f does not have to be resistant to all types of collisions, only to collisions of the type $f(x, y) = f(x, y^*)$. The best known collision attacks on hash functions such as MD5 and SHA-1 are multi-block collision attacks; here, first a pair of message blocks (m_1, m^*_1) with $h_1 = f(\text{iv}, m_1) \neq f(\text{iv}, m^*_1) = h^*_1$ is found, and then a new pair of message blocks (m_2, m^*_2) such that $f(h_1, m_2) = f(h^*_1, m^*_2)$ is found. Hence, the compression functions of MD5 and SHA-1 still resist collision attacks of the type considered in Theorem 3.1, and therefore, hash functions iterating these compression functions as in Construction 3.9 are (apparently) still collision resistant.

Another feature of the construction, which protects against shortcut attacks, is the fact that n bits out of the μ bits in the second input are fixed by the hash computation, and hence n degrees of freedom are lost for the attacker. The large number of degrees of freedom in MD4-style hash functions seems to have taken a major part in the success of collision attacks on these hash functions.

Performance

Assuming black-box access to the compression function, performance of the Knudsen-Thomsen construction compared to the Merkle-Damgård construction is reduced by a factor of $\mu/(\mu - n)$, since in the Merkle-Damgård construction, μ bits can be processed per compression function call, whereas in Construction 3.9, only $\mu - n$ bits can be processed for each call. For instance, if the compression function of SHA-1 (having $\mu = 512$ and $n = 160$) is employed in the Knudsen-Thomsen mode, then the speed of the hash function is expected to be reduced by a factor about 1.45 compared to "native" SHA-1.

Generic attacks

We investigate the effect on generic attacks of switching from the Merkle-Damgård construction to Construction 3.9. For descriptions of the generic attacks, see Section 2.2.

Since the last block, the length padding block, is processed in a different way than the actual message blocks (the first input is 0, whereas for actual message blocks it is at least 1), Construction 3.9 protects against the length extension attack. However, given a collision (M, M^*), where this collision occurs before the length padding block is processed (and $|M| = |M^*|$), such a collision may still be extended by appending any suffix y to both messages.

The complexity of finding multi-collisions in an iterated hash function depends solely on the amount of data passed on from one iteration to the next. If a collision spanning the whole data can be found, then the multi-collision attack can be launched. Therefore, Construction 3.9 offers no additional protection against *birthday* multi-collision attacks. However, since we claim that shortcut collision attacks are hindered, multi-collision attacks based on shortcut attacks are also claimed to be harder to carry out.

An expandable message is useful in the second preimage attack (Section 2.2.4) on the Merkle-Damgård construction. However, on Construction 3.9, substituting a message of length a for a message of length $b \neq a$ does not seem viable due to the counter. Constructing the expandable message in the first place is complicated. Therefore we believe that Construction 3.9

protects against the second preimage attack of Kelsey and Schneier.

The Nostradamus attack is only affected by the fact that there is less freedom in the choice of message blocks. Hence, in some cases collisions (and also the linking block) may have to span several message blocks instead of just one, as described in Section 2.2.5. Otherwise, the attack has the same complexity as it has on the Merkle-Damgård construction.

3.3.2 The wide-pipe and the double-pipe constructions

As a method to protect the Merkle-Damgård construction against generic attacks, as well as to allow "slight failures" in the compression function, Lucks proposed the wide-pipe and the double-pipe constructions [121]. These constructions operate with an internal state that is larger than the output. In this section, we denote by w the size of the internal state. In the wide-pipe construction, $w > n$, and in the double-pipe construction, $w = 2n$.

The wide-pipe construction

The wide-pipe construction is simply an enlarged Merkle-Damgård construction with a w-bit internal state size, and with the addition of an output transformation $\Omega : \{0,1\}^w \to \{0,1\}^n$. As above, we denote by f the compression function, now mapping as $\{0,1\}^w \times \{0,1\}^\mu \to \{0,1\}^w$. See Figure 3.5. Assuming that f and $\Omega \circ f$ are collision resistant, the multi-collision attack

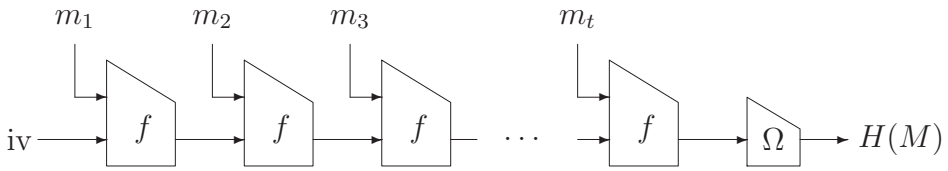


Figure 3.5: The wide-pipe construction.

of Joux (Section 2.2.3) now has complexity $t2^{w/2}$ for a 2^t-collision. With $w \geq 2n$, this is no better than a brute force multi-collision attack. The second preimage attack (Section 2.2.4) has complexity around $2^{w/2} + 2^{w-k}$, and hence, again with $w \geq 2n$, is no better than the brute force second preimage attack.

A shortcut collision attack on the compression function of complexity below $2^{w/2}$ may not pose a threat (with respect to collisions) on the full hash function, since $w > n$. Hence, this construction allows the underlying compression function to fail "slightly".

The double-pipe construction

The double-pipe construction uses two n-bit compression functions per message block. In effect, two chains are maintained during the processing of a message, but the chaining value of each chain is fed to the other chain, so that the compression function f must accept two n-bit chaining values and a message block. The last message block is only fed to one of the two chains, and the output of this chain is the output of the hash function. The cost in terms of efficiency compared to the Merkle-Damgård iteration of f is that now, f has to be applied twice as often, and in addition, the size of each message block decreases by n bits, due to the interchange of chaining values between the two chains. See Figure 3.6. Clearly, the two initial values for the

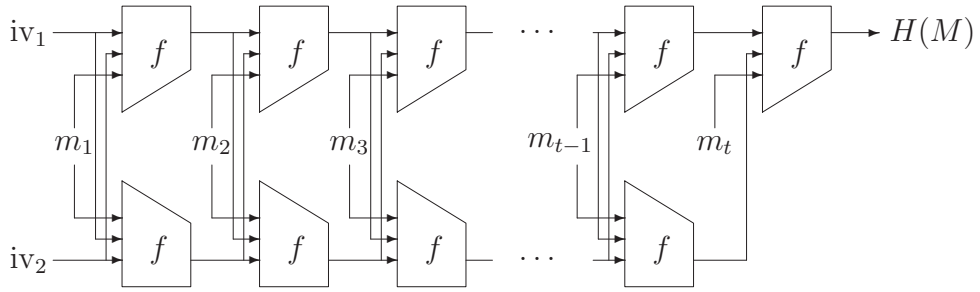


Figure 3.6: The double-pipe construction.

two chains must be different, since otherwise the two chains would contain identical chaining values.

When f is modelled as a random oracle, then the double-pipe construction resists collision, multi-collision, preimage and second preimage attacks.

3.3.3 Checksum-based hash functions

A checksum-based hash function is similar to the double-pipe construction, but there is no interchange of chaining values between the two chains, and one of the chains, the *checksum chain*, is often much more efficient than the other. The final output of the checksum chain is, in effect, processed as a last message block at the end. In other words, let $c : \{0,1\}^\mu \times \{0,1\}^\mu \rightarrow \{0,1\}^\mu$ be an efficient function that accepts a checksum state of size μ bits and a message block of size μ bits, and computes a new checksum state of size μ bits. c is iterated to form C in the same way that f is iterated to form F (see Section 2.1). Hence, C accepts inputs that are any multiple of μ bits in length. An initial checksum value must be defined. We shall generally denote the intermediate checksum values by v_i, and the initial one by v_0.

The checksum-based construction (as we define it here) is the construction that is identical to the Merkle-Damgård construction, except that the μ-bit string $C(\text{pad}^*_{\mu,\tau}(M))$ is appended to the padded message $\text{pad}^*_{\mu,\tau}(M)$ before hashing. We call the resulting hash function $\tilde{H}$. See Figure 3.7. An example

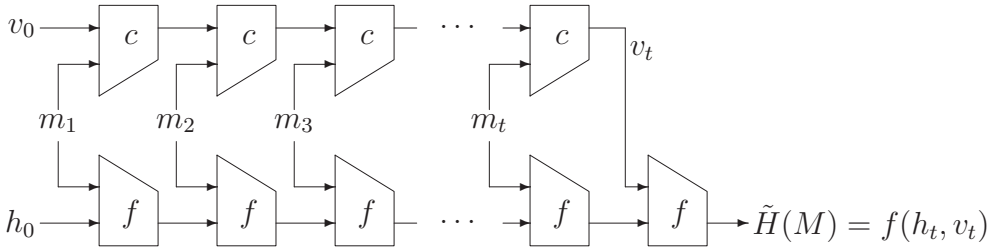


Figure 3.7: The Merkle-Damgård construction extended with a checksum.

of such a construction is the MD2 hash function [91] (see also Section 3.4.1), which has a non-linear, but invertible checksum function. Another example is 3C [72], which has (roughly) $c(v, m) = f(v, m) \oplus v$ and $v_0 = h_0$. Hence, the checksum is the XOR of all intermediate hash states. We describe generic attacks on checksum-based hash functions in Section 4.4.

Since appending a checksum is often thought of as an inexpensive way of increasing the security of a hash function, the checksum function must be efficient – preferably more efficient than the compression function of the hash function. This means that often the checksum function is simple, it may even be linear (corresponding to a modular addition or an XOR [72, 180]), or more generally, invertible (e.g., the checksum function of MD2). As in 3C, using $f(h, m)$ inside c can be done for free, since $f(h, m)$ is already computed in the "hash chain".

An alternative is to compute the hash of the message using two different hash functions, and then merge the two chains by processing the output of one hash function by the other. As an example, one may consider SHA-256 strengthened by a checksum computed using MD5. Although the output size of MD5 is less than the message block size of SHA-256, padding can be used to fill the final message block. The cost in terms of performance would be by a factor less than two, since most implementations of MD5 are (much) faster than comparable implementations of SHA-256.

Generic attacks apply to many checksum-based constructions, see [69], and Section 4.4 of this thesis. However, the added protection against shortcut attacks might be significant. Another advantage of checksum-based hash functions is that it allows the original hash function to be called in black-box mode, meaning no direct change in the original hash function is required,

only a type of message pre-processing (which can be done in parallel with the hash computation).

3.3.4 Multi-property preserving constructions

As mentioned in Section 2.2, the Merkle-Damgård construction preserves the collision resistance of the underlying compression function, but it does not preserve many other properties such as preimage and second preimage resistance, randomness etc. Hash functions are often used in applications that require stronger security notions than collision resistance. For instance, hash functions are widely used as instantiations of random oracles, and in order to justify this use, the hash construction should, preferably, preserve notions of pseudo-randomness. Hence, attempts have been made to design constructions that preserve additional properties of the underlying compression function.

One of the first discussions along the lines of multi-property preservation can be found in a Crypto '05 paper by Coron, Dodis, Malinaud, and Puniya [37]. Here, the problem of devising a hash function construction that preserves *indifferentiability* from a random oracle is considered. If the function f is indifferentiable from a random oracle $\mathcal{R}$ truncated to the same size as f, then using f instead of $\mathcal{R}$ in any protocol results in a protocol that is at least as secure as it was when using $\mathcal{R}$. Coron et al. show that the Merkle-Damgård construction does not preserve indifferentiability from a random oracle (from now on "the pseudorandom oracle property") of the underlying compression function, and a number of constructions that do are presented. One of these is the Merkle-Damgård construction with *prefix-free padding*. A padding function pad is prefix-free if for every pair (x, y) such that $x \neq y$, $\text{pad}(x)$ is not a prefix of $\text{pad}(y)$ (this does not hold for the padding functions introduced in Section 2.1).

Chang, Lee, Nandi, and Yung investigated [32] the pseudorandom oracle property preservation of a number of hash function constructions, including the block cipher based constructions from [163] and the MDC-2 construction (Section 3.1.2). Sixteen of the block cipher based constructions from [163] are shown to preserve the pseudorandom oracle property if a prefix-free padding function is applied to the message before hashing. The MDC-2 construction is shown *not* to preserve the pseudorandom oracle property.

Bellare and Ristenpart [8] showed that the hash function constructions proposed by Coron et al. in [37] are not collision resistance preserving. Hence, in some sense, the constructions are worse than Merkle-Damgård; for instance, cryptographic protocols proven secure in the random oracle model are provably secure if the compression function of the hash function preserves the

pseudorandom oracle property, but cryptographic protocols proven secure in the standard model are *not* provably secure if the compression function of the hash function is only collision resistant. Bellare and Ristenpart devise the so-called *enveloped Merkle-Damgård* (EMD) construction, which provably preserves collision resistance, the pseudorandom oracle property, and the pseudorandom function property. A pseudorandom function (PRF) is a function that accepts a key input, such that for every key, the function (computationally) emulates a random oracle with the same output size. We briefly describe the EMD construction. Given a compression function f, the message M is padded as in the Merkle-Damgård construction (Definition 2.2), except that the number of '0' bits is chosen such that the total length of the padded message is n bits short of a multiple of μ bits. Denote this padding function by pad_{EMD}; let $\text{pad}_{\text{EMD}}(M) = m_1 \| \cdots \| m_{t-1} \| m_t$, where $|m_i| = \mu$ for $1 \leq i \leq t-1$, and $|m_t| = \mu - n$. Define two distinct initial values, iv_1 and iv_2. Compute $h_{t-1} = F(\text{iv}_1, m_1 \| \cdots \| m_{t-1})$, and let the output of the hash function be $H(M) = f(\text{iv}_2, h_{t-1} \| m_t)$. See Figure 3.8.

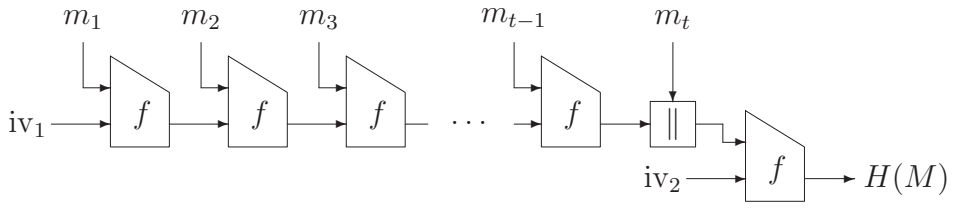


Figure 3.8: The Enveloped Merkle-Damgård (EMD) construction.

The EMD construction does not preserve all seven security notions of [176] (see also Section 1.1.7). This was shown by Andreeva, Neven, Preneel, and Shrimpton [5]. They propose the *random oracle XOR* (ROX) construction, which provably preserves all seven security notions (however, the pseudorandom oracle and pseudorandom function properties are not proven to be preserved). ROX is, to the best of our knowledge, the only hash function construction that provably preserves all seven security notions – however, the construction needs to perform a logarithmic number of calls to "small" random oracles, and therefore it is not as practical as the EMD construction. On the other hand, the random oracles may be instantiated by reasonably inefficient primitives due to the infrequent access required.

3.3.5 The sponge construction

Bertoni, Daemen, Peeters, and Van Assche introduced the *sponge construction* in [13]. A sponge is an object that digests a message in an iterated

fashion (the *absorbing phase*), and then outputs an infinite string in a similar way (the *squeezing phase*). When used in a practical application, the output of the sponge will be truncated to n bits.

Here we give a slightly simplified description of the sponge. Let f be a function accepting inputs of length ℓ bits, and returning outputs of the same size. Let S be a state of size ℓ bits, that is split into two parts; S_{i} and S_{p} of sizes μ and ν bits, respectively. We have $\ell = \mu + \nu$. The message M is padded and split into μ-bit blocks m_i, $0 < i \leq t$. An initial state S_0 is defined. The message M is then processed as follows:

$$S_i \leftarrow f(m_i \circ S_{\mathrm{i},i-1} \| S_{\mathrm{p},i-1}),$$

where $S_i = S_{\mathrm{i},i} \| S_{\mathrm{p},i}$, and '$\circ$' is some binary operation on elements of the set $\{0,1\}^{\mu}$. Simple examples are $a \circ b = a \oplus b$ (the message block is XORed to part of the state), and $a \circ b = a$ (the message block overwrites part of the state).
Once the message has been digested, the squeezing (output) phase starts. The output is formed in chunks z_i of $\kappa \leq \ell$ bits each as follows.

$$z_i \leftarrow \mathsf{trunc}_{\kappa}(S_{t+i-1}),$$

followed by

$$S_{t+i} \leftarrow f(S_{t+i-1}),$$

for $i \geq 1$. We note that padding of the message may ensure that a few "blank" applications of f are introduced before the output phase.

The function f may be modelled as a random function or a random permutation. In the former case, the resulting sponge is called a T-sponge. In the latter case it is called a P-sponge. The security of the sponge construction depends on the type of the sponge.

The security of a sponge construction can be expressed in terms of its *capacity* c, where $c \leq \nu$. The capacity is a measure of the strength of the function f. Here we assume that the output of the sponge is truncated to n bits. In terms of collisions, T-sponges and P-sponges provide the same amount of resistance. If the sponge has capacity c, then it is collision resistant up to a level of $\min(2^{c/2}, 2^{n/2})$. Preimage resistance depends on the type of sponge; a T-sponge of capacity c has preimage resistance $\min(2^c, 2^n)$, and a P-sponge of the same capacity has preimage resistance $\min(2^{c/2}, 2^n)$. A P-sponge of capacity c has second preimage resistance equal to its preimage resistance. The second preimage resistance of a T-sponge depends on the length L of the first preimage. For a T-sponge of capacity c, resistance is at a level of $\min(2^c/L, 2^n)$.

In [14], it is shown that the sponge construction is indifferentiable from a random oracle assuming that f is a random function or a random permutation, and that $c \geq 2n$.

Examples of hash functions that may be seen as sponge functions are Panama [40], RadioGatún [12], Keccak [15], Grindahl (Section 3.4.3), and Anaconda (Section 3.4.5). These are all P-sponges, illustrating that it is generally believed that more is known about building strong permutations than about building functions behaving randomly.

3.4 Dedicated designs

Dedicated hash functions are designs that were never intended to be used for other purposes than hashing. A dedicated hash function is often more efficient than a hash function based on an existing primitive such as a block cipher or a permutation. In practice, most hash functions in use are dedicated designs for this reason, but also because dedicated hash functions are not covered by export restrictions in most countries, whereas block ciphers may be.

One of the first dedicated hash functions was MD2, designed by Rivest in 1988. MD2 was targeted for 8-bit processors, and hence it was inefficient on the 32-bit processors that appeared in PCs in the last half of the 80s. MD4 was a dedicated 32-bit design that gained a large amount of popularity. A number of hash functions have been designed with MD4 as the main source of inspiration. These hash functions are often collectively termed the MD4 family. They are all based on the Merkle-Damgård construction.

In this section, we describe MD2 and the MD4 family, and we also describe three hash functions that have very little in common with MD4, and which are co-designed by the author: Grindahl, Dakota, and Anaconda.

3.4.1 MD2

MD2 is a hash function developed by Ronald L. Rivest in (or no later than) 1988, and published in 1989 [91, 120]. MD2 accepts messages of length any integral number of bytes, and returns a 128-bit hash. It is an iterated hash function, but not in Merkle-Damgård mode. The compression function is designed in a byte-oriented fashion, which makes it less suitable for implementation on modern processors such as the x86 family. Cryptanalysis of MD2 is described in Section 4.2.

Overall design

The input message M is padded so that its length becomes a multiple of 16 bytes. Padding is described below. The message is then split into t blocks $m_1, m_2, \ldots, m_t$ of 16 bytes each, and a 16-byte checksum block c is computed from the padded message. c is appended to the message as the $(t+1)$-th message block. The $t+1$ blocks are then processed sequentially as in the Merkle-Damgård construction. The initial state h_0 is the all zero 16-byte string.

Padding

If the original message consists of r bytes, then d bytes each having the value d are appended to the message, where d is the integer between 1 and 16 such that $r+d$ is a multiple of 16. Hence, all messages are padded, even if r is itself a multiple of 16. This padding function ensures that there is a one-to-one relationship between the original message and the padded message.

However, the padding function does *not* support a security proof as in the Merkle-Damgård construction, extending collision resistance of the compression function to collision resistance of the hash function. To see why, let M be a 15-byte message, and let M^+ be the padded version of M, i.e., the 16-byte message consisting of the 15 bytes of M and a '1'-byte. Assume that $f(h_0, M^+) = h_0$, and that the checksums of the two messages M^+ and $M^+\|M^+$ are identical. Then M and $M^+\|M$ collide in MD2, but no collision of the compression function f has been found. Of course, no message M with these properties is guaranteed to exist, but the technique can be generalised so that M may be several blocks in length, introducing additional degrees of freedom.

The compression function

In the following, we denote by x^b the b-th byte of a string x. The compression function $f : \{0,1\}^{128} \times \{0,1\}^{128} \to \{0,1\}^{128}$ works as follows. Given 16-byte strings h (the chaining input) and m (the message block), let

$$\begin{aligned} A_0 &\leftarrow h \\ B_0 &\leftarrow m \\ C_0 &\leftarrow h \oplus m. \end{aligned}$$

The concatenation $A_i\|B_i\|C_i$ (for arbitrary i) may be viewed as a single 48-byte entity X_i. To generate the output of the compression function, do the following (T is a temporary variable of length 1 byte).

$T \leftarrow 0$
for $i = 1$ to 18 **do**
  for $j = 0$ to 47 **do**
    $X_i^j \leftarrow S(T) \oplus X_{i-1}^j$
    $T \leftarrow X_i^j$
  end for
  $T \leftarrow T + i - 1 \bmod 256$
end for
return A_{18} {the first 16 bytes of X_{18}}

Here, S is an 8-bit *S-box*, the specification of which can be found below. Since only part of X_{18} is returned, the computation of the last 32 bytes is omitted in practice.

The checksum function

The checksum function c takes a 128-bit checksum state v (the initial state is the all-zero string) and a message block m, and produces a new 128-bit checksum state. The checksum function is invertible: given y and z, x can be efficiently found such that $c(x, y) = z$. Also, y can be efficiently computed from x and z. By efficient, we mean that it takes the same amount of time as evaluating c in the forward direction.

Let v denote the input checksum state, and let v^i be the ith byte of v. m is the message block with ith byte m^i. The state v is updated by m as follows (T is again a temporary variable of length 1 byte).

$T \leftarrow v^{15}$
for $i = 0$ to 15 **do**
  $v^i \leftarrow v^i \oplus S(T \oplus m^i)$
  $T \leftarrow v^i$
end for

Note the similarity between the checksum function and the compression function of MD2. Evaluating the MD2 checksum function corresponds to about $16/832 \approx 2^{-5.7}$ evaluations of the MD2 compression function (since 16 bytes are computed in the checksum function, and 832 bytes are computed in the compression function).

The S-box

The MD2 S-box S is specified in Table 3.1. This S-box is derived from the digits of the fractional part of π.

Table 3.1: The MD2 S-box.

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	29	2e	43	c9	a2	d8	7c	01	3d	36	54	a1	ec	f0	06	13
10	62	a7	05	f3	c0	c7	73	8c	98	93	2b	d9	bc	4c	82	ca
20	1e	9b	57	3c	fd	d4	e0	16	67	42	6f	18	8a	17	e5	12
30	be	4e	c4	d6	da	9e	de	49	a0	fb	f5	8e	bb	2f	ee	7a
40	a9	68	79	91	15	b2	07	3f	94	c2	10	89	0b	22	5f	21
50	80	7f	5d	9a	5a	90	32	27	35	3e	cc	e7	bf	f7	97	03
60	ff	19	30	b3	48	a5	b5	d1	d7	5e	92	2a	ac	56	aa	c6
70	4f	b8	38	d2	96	a4	7d	b6	76	fc	6b	e2	9c	74	04	f1
80	45	9d	70	59	64	71	87	20	86	5b	cf	65	e6	2d	a8	02
90	1b	60	25	ad	ae	b0	b9	f6	1c	46	61	69	34	40	7e	0f
a0	55	47	a3	23	dd	51	af	3a	c3	5c	f9	ce	ba	c5	ea	26
b0	2c	53	0d	6e	85	28	84	09	d3	df	cd	f4	41	81	4d	52
c0	6a	dc	37	c8	6c	c1	ab	fa	24	e1	7b	08	0c	bd	b1	4a
d0	78	88	95	8b	e3	63	e8	6d	e9	cb	d5	fe	3b	00	1d	39
e0	f2	ef	b7	0e	66	58	d0	e4	a6	77	72	f8	eb	75	4b	0a
f0	31	44	50	b4	8f	ed	1f	1a	db	99	8d	33	9f	11	83	14

3.4.2 The MD4 family

The title of this section refers to an ill-defined set of cryptographic hash functions. MD4, developed by Rivest in 1990, gave name to this "family" because it was the first hash function of its kind. The other members of the family were, at least to some extent, inspired by MD4. These include such hash functions as MD5, SHA-0, HAVAL [227], SHA-1, SHA-2, and others.

It can be difficult to pin-point the exact characteristics of hash functions in the MD4 family. Here are some suggestions.

- They are built upon (variants of) the Merkle-Damgård construction
- The compression function can often be seen as a block cipher in Davies-Meyer mode (hence, the message block is the key, and the chaining input is the plaintext block of the block cipher) – we stress, however, that this "block cipher" was in all cases designed specifically for the hash function in question
- The compression function consists of a relatively large number of simple steps, that each update one or a few *registers* in a state, that contains from four to about eight registers
- Each register is a 32-bit or a 64-bit value

- There is a relatively simple *message expansion*, or, when the compression function is viewed as a block cipher in Davies-Meyer mode, *key schedule*.

Since the MD4 family of hash functions has played an extremely important role in the world of cryptographic hash functions, we briefly describe some of the members here.

MD4

MD4 is a 128-bit hash function in Merkle-Damgård mode. The message is padded according to Definition 2.2, with $\mu = 512$ and $\tau = 64$. The initial value is (in hexadecimal)

$$\text{iv} = \mathsf{67452301\ efcdab89\ 98badcfe\ 10325476}.$$

MD4 assumes a little-endian byte ordering (hence, the sequence 00 01 02 03 of bytes, in hexadecimal, will be read as a single 32-bit word 03020100).

The 128-bit chaining input h to the compression function is copied into four 32-bit state registers, a, b, c, and d. These registers are updated via 48 steps (separated into three rounds of 16 steps). The 512-bit message is expanded into 48 words of 32 bits, each word affecting one of the 48 steps. After the 48 steps, the registers are added (modulo 2^{32}) to the four words of the chaining input (this is the feed-forward in the Davies-Meyer construction), and the sum is returned as the output of the compression function.

Each step in fact only updates one of the registers via a step update function. The step update function takes the four registers, a message word, and a 32-bit constant (which changes in every round) as input, and outputs a single 32-bit word. The step update function changes slightly for each step. It consists of modulo 2^{32} additions, a rotation, and a (bitwise) Boolean function which changes for every round (hence, there are three different Boolean functions).

The Boolean functions all accept three 32-bit inputs, and produce, in a balanced way, a single 32-bit output. Denote by ℓ_j the Boolean function used in round j (counting starts from 0). The three Boolean functions are defined as follows.

$$\begin{aligned}
\ell_0(X,Y,Z) &= (X \wedge Y) \vee (\neg X \wedge Z) \\
\ell_1(X,Y,Z) &= (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z) \\
\ell_2(X,Y,Z) &= X \oplus Y \oplus Z.
\end{aligned}$$

Here (and in the following), '$\wedge$' means logical AND, '$\vee$' means logical OR, and '$\neg$' means logical negation.

The 32-bit constants k_j used are distinct for each round j. They are defined as follows (written in hexadecimal).

$$\begin{aligned} k_0 &= \mathsf{0} \\ k_1 &= \mathsf{5a827999} \\ k_2 &= \mathsf{6ed9eba1}. \end{aligned}$$

The message expansion is simply a repetition of the 16 message words of 32 bits each, where in the second and the third rounds the message words appear in a different order. Let u_i be the ith message word, $0 \leq i < 16$. Then, in round 0, the message words appear in the order $u_0, u_1, \ldots, u_{15}$. In round 1, the message words appear in the order

$$u_0, u_4, u_8, u_{12}, u_1, u_5, u_9, u_{13}, u_2, u_6, u_{10}, u_{14}, u_3, u_7, u_{11}, u_{15}.$$

In round 2, the message words appear in the order

$$u_0, u_8, u_4, u_{12}, u_2, u_{10}, u_6, u_{14}, u_1, u_9, u_5, u_{13}, u_3, u_{11}, u_7, u_{15}.$$

We refer to the message word used in step j as w_j.

A rotation means a cyclic shift of the bits in a word, and rotating the value x by s positions to the left is written $x^{\lll s}$. In each step, a left-rotation takes place. The number of positions rotated left is $3, 7, 11, 19, \ldots$ (recurring) in round 0, $3, 5, 9, 13, \ldots$ (recurring) in round 1, and $3, 9, 11, 15, \ldots$ (recurring) in round 2.

The step update function updates the four registers a, b, c, d to a', b', c', d' as follows (where we omit indices referring to the round or step number, and additions are to be taken modulo 2^{32}):

$$\begin{aligned} b' &\leftarrow (a + \ell(b, c, d) + w + k)^{\lll s} \\ c' &\leftarrow b \\ d' &\leftarrow c \\ a' &\leftarrow d. \end{aligned}$$

The copying of registers is, in practice, done implicitly.

MD4 is a very fast hash function. Unfortunately, it also turned out to be rather insecure. Cryptanalysis of MD4 is carried out, improved, described, or otherwise treated in a number of papers, e.g., [54, 59, 61, 62, 118, 119, 138, 188–190, 192, 211, 214, 223]. The best known collision attack on MD4 finds collisions in a tiny fraction of a second on a standard PC.

MD5

Weaknesses of MD4 were found rather quickly after its inception. This lead to the development of MD5, which is an improved variant of MD4. However, an early paper [55], containing a collision attack on the compression function of MD5, doubted the amount of improvement provided by MD5 over MD4. Still, MD5 uses four rounds instead of three, unique constants for each step, and a new Boolean function, and it resisted devastating attacks much longer than MD4.

The initial value, the padding, the byte ordering, and the feed-forward of the chaining input are all inherited from MD4.

As in MD4, the message expansion is a repetition of the 16 message words of 32 bits. In round 0, the message words come in the same order as in the message itself. In rounds 1–3, the ordering is different; in round 1, the message word used in step j is word no. $(5j + 1) \bmod 16$, in round 2 it is word no. $(3j + 5) \bmod 16$, and in round 3 it is word no. $7j \bmod 16$.

The rotation values are also different. In round 0 they are $7, 12, 17, 22, \ldots$ (recurring), in round 1 they are $5, 9, 14, 20, \ldots$ (recurring), in round 2 they are $4, 11, 16, 23, \ldots$ (recurring), and in round 3 they are $6, 10, 15, 21, \ldots$ (recurring).

The constant used in step j is the absolute value of $\sin(j + 1) \times 2^{32}$, rounded down (towards zero). The Boolean functions ℓ_j (j representing the round number) are

$$\begin{aligned}
\ell_0(X, Y, Z) &= (X \wedge Y) \vee (\neg X \wedge Z) \\
\ell_1(X, Y, Z) &= (X \wedge Z) \vee (Y \wedge \neg Z) \\
\ell_2(X, Y, Z) &= X \oplus Y \oplus Z \\
\ell_3(X, Y, Z) &= Y \oplus (X \vee \neg Z).
\end{aligned}$$

The step update function is also slightly different. In each step, the following is performed.

$$\begin{aligned}
b' &\leftarrow b + (a + \ell(b, c, d) + w + k)^{\lll s} \\
c' &\leftarrow b \\
d' &\leftarrow c \\
a' &\leftarrow d.
\end{aligned}$$

MD5 is about 33% slower than MD4, because one round is added. As mentioned, a collision was found in the MD5 compression function soon after it was developed, but it took quite a few years before an attack on the full hash function appeared. The following papers contain cryptanalytic results

on MD5: [11, 22, 55, 60, 118, 187–189, 191, 198, 199, 205, 216]. The currently best collision attack on MD5 finds collisions in a few seconds.

SHA-0

The U.S. standardisation body NIST published the *Secure Hash Standard* in May 1993. The hash function underlying the standard was called the *Secure Hash Algorithm* (SHA); today it is more commonly referred to as SHA-0.

SHA-0 is also built upon the same principles as MD4. However, SHA-0 is a 160-bit hash function, keeping five registers in the state: a, b, c, d, and e. The initial value is

$$\text{iv} = \mathsf{67452301\ efcdab89\ 98badcfe\ 10325476\ c3d2e1f0}.$$

Padding is the same as in MD4, and the chaining input is fed forward in the same way, but all SHA functions assume a big-endian byte ordering (hence, the sequence 00 01 02 03 of bytes will be read as a single 32-bit word 00010203).

The message expansion is somewhat more complicated than in MD4 and MD5, and there are four rounds of 20 steps each. The four Boolean functions are defined as follows.

$$\begin{aligned}
\ell_0(X,Y,Z) &= (X \wedge Y) \vee (\neg X \wedge Z) \\
\ell_1(X,Y,Z) &= X \oplus Y \oplus Z \\
\ell_2(X,Y,Z) &= (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z) \\
\ell_3(X,Y,Z) &= X \oplus Y \oplus Z.
\end{aligned}$$

Notice that $\ell_1 = \ell_3$.

There are four constants, one for each round. These are:

$$\begin{aligned}
k_0 &= \mathsf{5a827999} \\
k_1 &= \mathsf{6ed9eba1} \\
k_2 &= \mathsf{8f1bbcdc} \\
k_3 &= \mathsf{ca62c1d6}.
\end{aligned}$$

The message expansion works as follows. Let $u_0, u_1, \ldots, u_{15}$ be the 16 input words of 32 bits. The 80 words w_i, $0 \leq i < 80$, in the expanded message are defined as follows.

$$w_i = \begin{cases} u_i & \text{for } 0 \leq i < 16 \\ u_{i-3} \oplus u_{i-8} \oplus u_{i-14} \oplus u_{i-16} & \text{for } 16 \leq i < 80 \end{cases}$$

A SHA-0 step consists of the following operations.

$$\begin{aligned} a' &\leftarrow a^{\lll 5} + \ell(b,c,d) + e + w + k \\ b' &\leftarrow a \\ c' &\leftarrow b^{\lll 30} \\ d' &\leftarrow c \\ e' &\leftarrow d. \end{aligned}$$

SHA-0 was withdrawn by the NSA shortly after publication. No explanations for this withdrawal were given. A hint towards the reasons for the withdrawal came in 1998 with a paper [31] by Chabaud and Joux, describing a collision attack on SHA-0. Other cryptanalytic results on SHA-0 appear in [16, 17, 30, 139, 217]. One of only a few published implementations of collision attacks on SHA-0 was developed by the author and can be found at [203]; this implementation is based on the attack described in [217].

SHA-1

SHA-0 was replaced by SHA-1 in 1995. SHA-1 is arguably the most widely deployed cryptographic hash function. It appears in a huge number of cryptographic standards, protocols, schemes etc.

SHA-1 is a minor modification of SHA-0; the only difference lies in the message expansion, which, using the same terminology as above, is described as

$$w_i = \begin{cases} u_i & \text{for } 0 \leq i < 16 \\ (u_{i-3} \oplus u_{i-8} \oplus u_{i-14} \oplus u_{i-16})^{\lll 1} & \text{for } 16 \leq i < 80 \end{cases}$$

This minor modification bought SHA-1 some 7 years in terms of cryptanalysis, compared to SHA-0. The first collision attack (of complexity about 2^{69}) on SHA-1 appeared in 2005, together with a series of collision attacks on other hash functions with Xiaoyun Wang as the dominating character. Cryptanalytic results and other observations on SHA-1 appear in [17, 30, 51, 52, 80, 129, 158, 159, 169, 201, 215, 221].

SHA-2

With the publication of the *Advanced Encryption Standard* [145] in 2001, new hash functions with larger output sizes were needed to suit the key sizes in the AES. This led to the development of three new hash functions, SHA-256, SHA-384, and SHA-512, collectively termed SHA-2, published in 2002 [146]. The expected collision resistance of these hash functions matches the three key lengths of the AES. In 2004, an additional hash function, SHA-224, was

added to the SHA-2 family [148]. Its expected collision resistance matches the key length of two-key TDEA [152].

The SHA-2 hash functions are somewhat more involved than the previously described hash functions. Most importantly, the message expansion is much more complicated, and two registers are updated in each step. SHA-224 and SHA-256 are constructed in the same way, but they use different initial values, and in SHA-224, a 256-bit state is truncated to 224 bits in the end. Similar differences appear between SHA-384 and SHA-512.

SHA-256 and SHA-512 differ in the word size; SHA-256 uses 32-bit words, and SHA-512 uses 64-bit words. The number of steps is 64 in SHA-256 and 80 in SHA-512. Apart from this, there are only minor differences. Here, we shall only describe SHA-256 in detail.

SHA-256 uses the same padding technique as SHA-1, and the compression function is also a block cipher in Davies-Meyer mode. The initial value of SHA-256 is

$$\begin{aligned} \text{iv} &= \texttt{6a09e667 bb67ae85 3c6ef372 a54ff53a} \\ &\quad \texttt{510e527f 9b05688c 1f83d9ab 5be0cd19}. \end{aligned}$$

SHA-256 uses a number of Boolean functions. First, the following two functions applied on a single 32-bit word are used in the message expansion.

$$\begin{aligned} \sigma_0^{\{256\}}(x) &= x^{\ggg 7} \oplus x^{\ggg 18} \oplus x^{\gg 3} \\ \sigma_1^{\{256\}}(x) &= x^{\ggg 17} \oplus x^{\ggg 19} \oplus x^{\gg 10}. \end{aligned}$$

Here, $x^{\ggg s}$ means x rotated right by s positions (i.e., in the case of 32-bit words, $x^{\ggg s} = x^{\lll 32-s}$), and $x^{\gg s}$ means x shifted right by s positions (resulting in the most significant s bits of the result being zeroes).
Two other functions operating on a single 32-bit word are the following.

$$\begin{aligned} \Sigma_0^{\{256\}}(x) &= x^{\ggg 2} \oplus x^{\ggg 13} \oplus x^{\ggg 22} \\ \Sigma_1^{\{256\}}(x) &= x^{\ggg 6} \oplus x^{\ggg 11} \oplus x^{\ggg 25}. \end{aligned}$$

Moreover, ℓ_0 and ℓ_2 as defined in SHA-0 and SHA-1 are used. We use the same names here.

The SHA-256 compression function takes a 512-bit message (16 words $u_0, u_1, \ldots, u_{15}$ of 32 bits each) and expands it into a 2048-bit message (64 words $w_0, w_1, \ldots, w_{63}$ of 32 bits each) as follows.

$$w_i = \begin{cases} u_i & \text{for } 0 \le i < 16 \\ \sigma_1^{\{256\}}(u_{i-2}) + u_{i-7} + \sigma_0^{\{256\}}(u_{i-15}) + u_{i-16} & \text{for } 16 \le i < 64 \end{cases}$$

Additions are, again, to be taken modulo 2^{32} (notice the difference compared to SHA-0 and SHA-1, where XORs are used). It is clear that this message expansion introduces more diffusion than the ones appearing in SHA-0 and SHA-1.

SHA-256 maintains a state of eight registers, each being 32 bits in length. The state is updated through 64 steps. Each step is more complicated than the steps used in SHA-0 and SHA-1; two registers are updated via a function of a number of other registers. Each step involves a distinct constant k_j. The constants are the first 32 bits of the fractional parts of the cube roots of the first 64 prime numbers – from left to right:

428a2f98	71374491	b5c0fbcf	e9b5dba5	3956c25b	59f111f1	923f82a4	ab1c5ed5
d807aa98	12835b01	243185be	550c7dc3	72be5d74	80deb1fe	9bdc06a7	c19bf174
e49b69c1	efbe4786	0fc19dc6	240ca1cc	2de92c6f	4a7484aa	5cb0a9dc	76f988da
983e5152	a831c66d	b00327c8	bf597fc7	c6e00bf3	d5a79147	06ca6351	14292967
27b70a85	2e1b2138	4d2c6dfc	53380d13	650a7354	766a0abb	81c2c92e	92722c85
a2bfe8a1	a81a664b	c24b8b70	c76c51a3	d192e819	d6990624	f40e3585	106aa070
19a4c116	1e376c08	2748774c	34b0bcb5	391c0cb3	4ed8aa4a	5b9cca4f	682e6ff3
748f82ee	78a5636f	84c87814	8cc70208	90befffa	a4506ceb	bef9a3f7	c67178f2

In each step, the following operations are performed (tmp1 and tmp2 are temporary variables, and $a, b, \ldots, h$ are the eight registers of the state).

$$\begin{aligned}
\text{tmp1} &\leftarrow \Sigma_0^{\{256\}}(a) + \ell_2(a,b,c) \\
\text{tmp2} &\leftarrow \Sigma_1^{\{256\}}(e) + \ell_0(e,f,g) + h + w + k \\
a' &\leftarrow \text{tmp1} + \text{tmp2} \\
b' &\leftarrow a \\
c' &\leftarrow b \\
d' &\leftarrow c \\
e' &\leftarrow d + \text{tmp2} \\
f' &\leftarrow e \\
g' &\leftarrow f \\
h' &\leftarrow g.
\end{aligned}$$

SHA-256 (and the other SHA-2 hash functions) is still considered secure. It seems that especially the change in the message modification has made cryptanalysis on SHA-2 more difficult than on SHA-0 and SHA-1. Generally, the degree of non-linearity in the SHA-2 hash functions has been improved compared to the previous standards. Some cryptanalytic results on SHA-2 may be found in [74, 84, 128, 157, 184–186, 222]. Currently, the best collision attack on a reduced variant of SHA-256 covers 24 out of the 64 steps.

3.4.3 Grindahl

Grindahl is a hash function designed by Knudsen, Rechberger, and this author, and published at FSE 2007 [107]. It was broken by Peyrin later the same year [154] in the form of a collision attack being a factor 2^{16} faster than the birthday attack. Grindahl is in fact a collection of hash functions with two concrete instances, Grindahl-256 producing 256 bits of output, and Grindahl-512 producing 512 bits of output. The collision attack was only described for the shorter variant.

Grindahl employs an alternative design strategy for hash functions compared to the MD4 family. It is comparable to some of the designs of Daemen et al., such as SubHash and StepRightUp [39], Panama [40], and the more recent proposal RadioGatún [12]. All these hash functions can be described as (variants of) P-sponges (Section 3.3.5).

The Grindahl design

The overall design strategy is called "Concatenate-Permute-Truncate", but may also be described as a P-sponge. This design principle was first proposed by Merkle and used in his hash function Snefru [133], and it requires the existence of a non-linear permutation. Grindahl employs a highly parametrisable permutation, hence in effect a collection of non-linear permutations. These are described below. The general design and the permutations together form the Grindahl hash functions.

General strategy. Let P be an ℓ-bit permutation, with $\ell > n$. Let $\mu \leq \ell - n$ be the size of each message block. Define $c = \ell - \mu$. Let s_0 be an initial state of size c bits.

The "Concatenate-Permute-Truncate" principle is the following: let M be the message to be hashed, pad it to M^+, a multiple of μ bits in length, and split M^+ into t blocks $m_1, m_2, \ldots, m_t$ of μ bits each. Then, for $1 \leq i \leq t-1$ do

$$S_i \leftarrow m_i \| s_{i-1} \qquad \text{(Concatenate)} \qquad (3.2)$$

$$S_i^* \leftarrow P(S_i) \qquad \text{(Permute)} \qquad (3.3)$$

$$s_i \leftarrow \mathsf{trunc}_c(S_i^*) \qquad \text{(Truncate)} \qquad (3.4)$$

Hence, a message block is concatenated with the state to form what we shall call the *extended state*, on which some permutation P is applied. Subsequently, the extended state is truncated down to the new state. The steps (3.2)–(3.4) form an *input round*.

The above process digested all but the last message block m_t. This message block is processed as

$$S_t^* \leftarrow P(m_t \| s_{t-1}).$$

Hence, the state is not truncated after processing the last message block.

We define an output transformation consisting of a number ν of *blank rounds* and a truncation step at the end. Blank rounds are defined as follows. For $t < i \leq t + \nu$ do

$$S_i^* \leftarrow P(S_{i-1}^*) \ .$$

Finally, the output of the hash function is $\mathsf{trunc}_n(S_{t+\nu}^*)$.

Invertibility. Assuming that the permutation P is efficiently invertible, the hash function is not one-way in the sense that for a given output, some initial state and a message producing that output can easily be found. However, this does not directly give rise to proper preimage and second preimage attacks. If P is sufficiently "strong", then an attacker will have no control over the initial state obtained.

The success probability of meet-in-the-middle attacks (see Section 4.1.4) is affected by the value of c above. If no weaknesses of P are exploited, then internal collision attacks (collisions before the blank rounds) and meet-in-the-middle attacks have complexity $2^{c/2}$. If one requires that no preimage and second preimage attacks better than a brute force search exist, then one has to choose $c \geq 2n$.

Design approach for the permutation. A well-known family of permutations is the block cipher algorithm Rijndael [42], a subset of which was adopted as the Advanced Encryption Standard (AES) by the U.S. government in 2001 [145]. What follows is a design approach for the permutation P that closely follows the principles underlying Rijndael. The approach bears some resemblance to the leak-extraction method of the stream cipher LEX [20], and to the MAC construction ALRED [45].

Consider an extended state as a matrix of bytes. The matrix contains u rows and v columns (so that $\ell = 8uv$). In the following we assume that $v \geq u$. If the matrix is denoted A, then each element in the matrix can be indexed as $a_{i,j}$, meaning the element in row i, column j. Row indices are always assumed to be reduced modulo u, and column indices are assumed to be reduced modulo v. In the AES, $u = v = 4$.

We assume that μ is a multiple of 8, and we define $\lambda = \mu/8$ as the number of bytes in a message block. Hence, according to (3.4) and (3.2),

in the process of truncation followed by concatenation (with the following message block), λ extended state bytes are overwritten by a message block. Therefore, they do not have to be computed in all except the last input round.

We briefly describe three of the four transformations defined in the AES. More details can be found in the standard [145]. The four transformations are called SubBytes, ShiftRows, MixColumns, and AddRoundKey. We do not use AddRoundKey directly in this design. Instead, we introduce a related transformation, AddConstant.

SubBytes is a non-linear byte-wise substitution function. Hence, SubBytes substitutes each byte $a_{i,j}$ in the matrix by another byte $S(a_{i,j})$, where S is an S-box. The same S-box is used regardless of the values of u and v. The specification of this S-box can be found in [42, 145], or in Table 5.1 (page 136).

ShiftRows cyclically shifts bytes a number of positions to the right along each row in the matrix. To be more precise, let $\sigma = [\sigma_0, \sigma_1, \ldots, \sigma_{u-1}]$ be a list of distinct integers in the range from 0 to $v-1$. These are called the *shift values*. Then ShiftRows is defined as the transformation that maps the matrix A to A^* such that

$$a^*_{i,j} \leftarrow a_{i,j-\sigma_i} \quad \text{for } 0 \le i < u \text{ and } 0 \le j < v.$$

In the AES, $\sigma = [0, 3, 2, 1]$.

MixColumns mixes bytes within each column of the matrix. The transformation can be described as the matrix multiplication

$$A \leftarrow C \times A,$$

where C is a circulant matrix (meaning that each row is equal to the row above cyclically shifted by one position). The matrix multiplication takes place in the finite field $\mathbb{F}_{256}$ defined by the polynomial $x^8 \oplus x^4 \oplus x^3 \oplus x \oplus 1$ over $\mathbb{F}_2$. There is a direct mapping from elements of this field to bytes, and back. The matrix C is defined as follows in the AES:

$$C = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix},$$

where bytes are written in hexadecimal. When $u \neq 4$, the matrix C needs to be redefined. It is important that the error-correcting code with generator

matrix $[I\ C^{\mathrm{T}}]$, where I is the identity matrix and C^{T} means C transposed, is *maximum distance separable* (MDS, [123]), meaning that if two states differ in a column in $d > 0$ positions before MixColumns, then after MixColumns they differ in at least $u - d + 1$ positions.

AddConstant simply XORs a constant matrix onto the state matrix.

The four transformations operate on a matrix of bytes. Mapping a byte string to a matrix and back is done as follows. Let the byte string be x, where x_i denotes the ith byte, and counting starts from 0. Map x to the matrix A such that

$$A = \begin{bmatrix} x_0 & x_u & \cdots & x_{u(v-1)} \\ x_1 & x_{u+1} & \cdots & x_{u(v-1)+1} \\ \vdots & \vdots & \ddots & \vdots \\ x_{u-1} & x_{2u-1} & \cdots & x_{uv-1} \end{bmatrix}.$$

This mapping has a natural inverse. From now on, we shall not distinguish between byte strings (of the appropriate length) and matrices.

The names of the three transformations that we "borrow" from the AES refer to the definitions given in [145]. We shall denote variants tailored for Grindahl by adding a star '$\star$' next to the name. Given their appropriate definitions we define the permutation P as

$$P(x) = \mathsf{MixColumns}\star \circ \mathsf{ShiftRows}\star \circ \mathsf{SubBytes} \circ \mathsf{AddConstant}(x).$$

Birthday attacks. If P were an ideal permutation, then internal collisions, preimages and second preimages would have complexity $2^{c/2}$. However, P is obviously not ideal. In fact, it is easy to see that the complexity of, e.g., an internal collision attack for this choice of P is at most $2^{(c-\mu)/2}$: assume that the first column of the matrix is overwritten by the message input. Compute the extended state before the blank rounds of a number of different messages. Now append two constant blocks to all messages. The first constant message block overwrites the first column of the extended state. Then the permutation is applied, where ShiftRows$\star$ moves u bytes into the first column of the extended state, and subsequently MixColumns$\star$ mixes the bytes in the column. The second constant message block overwrites this column. This means that if two extended states agree on all bytes except the first column and the bytes that are moved into the first column by ShiftRows$\star$, then the two extended states will agree on all bytes after the second constant message block. The expected number of messages needed for this attack to succeed is $2^{(c-\mu)/2}$. In fact, this approach can be generalised to every way of mapping an input message to the extended state.

Since the permutation is invertible, the entire hash function is invertible, and hence meet-in-the-middle attacks in time $2^{(c-\mu)/2}$ can be applied (exploiting the same property as the collision attack described above).

Design parameters for the permutation

We now present some considerations with respect to the design parameters introduced above.

Shift values. The shift values used in ShiftRows$\star$ should be chosen carefully. Most importantly, they should ensure that the entire state depends on the message input as quickly as possible when considering that the first μ bytes of the state are overwritten by message input in every round. This means, for instance, that no shift value can be zero, and that all shift values should be distinct.

Several tuples of shift values ensure full diffusion after the same (minimum) number of rounds r. However, of these, some are better than others in the sense that a larger part of the state depends on every message byte after $1, 2, \ldots, r-1$ rounds.

State geometry. Choices for u and v are a trade-off between two distinct properties. If the two numbers are about the same, diffusion happens faster than if $v \gg u$. On the other hand, with a wider state and only a single column being used for message input, the birthday attack as described above has a higher complexity, and hence the extended state may only need to be slightly larger than the output. For implementation reasons, it makes sense to choose u to be a multiple of 4, since then, on a 32-bit machine, SubBytes and MixColumns$\star$ can be performed in one by table look-ups.

Design parameters for the output transformation

The number ν of blank rounds in the output transformation should be chosen such that the last message block affects all output bytes.

We note that with ν blank rounds, the actual number of invocations of P after the last message block is concatenated with the state is $\nu + 1$. During these rounds, no extended state bytes are overwritten by message input. Hence, if the entire extended state is affected by all bytes in the last message block after r rounds, then one should choose $\nu \geq r - 1$.

Grindahl-256

Grindahl-256 is defined as follows. The extended state matrix consists of $u = 4$ rows and $v = 13$ columns. Each message block is $\mu = 32$ bits (i.e., $\lambda = 4$ bytes) in length. The number of blank rounds in the output transformation is $\nu = 8$.

The shift values used in the ShiftRows$\star$ transformation are $[1, 2, 4, 10]$. For these shift values, every message byte affects the entire extended state after four rounds. SubBytes and MixColumns$\star$ are defined as in the AES, and AddConstant simply flips the least significant bit of the state.

Grindahl-512

Grindahl-512 maintains a state of $u = 8$ rows and $v = 13$ columns. Each message block is $\lambda = 8$ bytes long (i.e., $\mu = 64$). There are $\nu = 8$ blank rounds in the output transformation.

The shift values for ShiftRows$\star$ are $[1, 2, 3, 4, 5, 6, 7, 8]$, which cause full diffusion after three rounds. MixColumns$\star$ defines

$$C = \begin{bmatrix} 02 & 0c & 06 & 08 & 01 & 04 & 01 & 01 \\ 01 & 02 & 0c & 06 & 08 & 01 & 04 & 01 \\ 01 & 01 & 02 & 0c & 06 & 08 & 01 & 04 \\ 04 & 01 & 01 & 02 & 0c & 06 & 08 & 01 \\ 01 & 04 & 01 & 01 & 02 & 0c & 06 & 08 \\ 08 & 01 & 04 & 01 & 01 & 02 & 0c & 06 \\ 06 & 08 & 01 & 04 & 01 & 01 & 02 & 0c \\ 0c & 06 & 08 & 01 & 04 & 01 & 01 & 02 \end{bmatrix}.$$

This matrix ensures that MixColumns$\star$ has the MDS property mentioned previously. SubBytes is the same as in Rijndael, and AddConstant again flips the least significant bit of the state.

Padding

Padding for both Grindahl-256 and Grindahl-512 is performed as follows. Append a '1' bit to the message, and then enough '0' bits to fill the last message block. Finally, append a 64-bit representation of the number of message blocks in the padded message. This means that both hash functions can digest messages of size at most $2^{64} - 1$ message blocks including the padding itself. (Note the difference between this padding function and those of Definitions 2.1 and 2.2.)

Compression function mode

A compression function mode of Grindahl is proposed. The compression function is defined as the function that calls Grindahl-256/Grindahl-512 without padding, using a single 40-block message (i.e., a message of size 1280, respectively 2560 bits). The chaining input is part of this message, meaning that for Grindahl-256, the corresponding compression function f_{256} maps as $\{0,1\}^{256} \times \{0,1\}^{1024} \rightarrow \{0,1\}^{256}$. For Grindahl-512, the corresponding compression function f_{512} maps as $\{0,1\}^{512} \times \{0,1\}^{2048} \rightarrow \{0,1\}^{512}$. In both cases, the compression function requires 48 calls of the permutation P. An additional salt, key, counter, or randomisation input may be added, and this is proposed to simply be prepended the 40-block message.

Implementation aspects

Grindahl implementations may benefit from the large amount of implementation research already done on the AES.

We may compare the speed of Grindahl with the speed of the AES. Grindahl-256 computes 12 columns per message block of 32 bits (since the first column does not have to be computed for most message blocks). AES using 128-bit keys, in comparison, computes 40 columns per block of 128 bits, equivalent to 10 columns per 32 bits. There is also a difference between using AddRoundKey and AddConstant, since the constant used in Grindahl-256 contains zeroes in most positions. In summary, Grindahl-256 is expected to have similar running time as the AES, or to be up to about 20% slower. Grindahl-512 is intended for 64-bit processors, where its speed is expected to be similar to Grindahl-256. On 32-bit processors, Grindahl-512 is expected to be at least twice slower than Grindahl-256.

An implementation of Grindahl-256 in C, available from [77], reaches a speed of about 24 clock cycles per byte of data ("cycles/byte") on an Intel Core2 Duo (E4600). A preliminary implementation with some inline assembly goes below 20 cycles/byte. Comparing with the best AES implementations in CTR mode (e.g., those of the eSTREAM project [65]) reaching speeds of about 12 cycles/byte on the same processor, it seems that some improvements may be possible. However, in CTR mode, the first one or two rounds can be simplified since the "plaintext" only changes slightly. Therefore, comparisons are not trivial.

In hardware, implementations seem to benefit from the low memory requirements. Only $\ell \in \{416, 832\}$ bits of memory are required, whereas in, e.g., SHA-256, memory requirements are 1024 bits.

A collision attack on Grindahl-256

Although this chapter is a design chapter, and not a cryptanalysis chapter, we would like to briefly describe here some properties of Grindahl-256, together with a collision attack by Peyrin [154].

A differential attack [18, 19] considers differences between state values when processing two different messages with a given difference. An *active byte* is a byte containing a difference (a byte that is not active may be called *passive*). When an active byte goes through the S-box, it becomes difficult to predict the output difference. The best probability of a given input/output difference pair for the S-box used in the AES and Grindahl is 2^{-6} [42, 145].

An exhaustive search through all possible input difference and difference propagation patterns shows that Grindahl-256 has the following property: an internal collision for Grindahl-256 requires at least six input rounds, and moreover, any characteristic starting or ending in the extended state with no difference contains at least one round where at least half the state bytes are active.

The large number of active bytes for an internal collision to occur is expected to rule out classical differential attacks. However, an anonymous reviewer of the paper submitted to FSE 2007 described a potential attack method. Here, one does not take actual differences into account, but only considers bytes as either passive or active.

When considering only passive/active bytes, the S-box can be ignored in the analysis because a passive byte is always mapped to a passive byte, and an active byte is always mapped to an active byte. Instead, the MixColumns$\star$ transformation becomes "probabilistic", meaning that its effect is to some extent unpredictable. The MDS property of the transformation ensures that transitions from α active input bytes to β active output bytes in a column happen with the *approximate* probabilities found in Table 3.2. An attacker

Table 3.2: Approximate transition probabilities through MixColumns (as defined in the AES) for α active input bytes to β active output bytes.

		β				
		0	1	2	3	4
	0	1	0	0	0	0
	1	0	0	0	0	1
α	2	0	0	0	2^{-8}	1
	3	0	0	2^{-16}	2^{-8}	1
	4	0	2^{-24}	2^{-16}	2^{-8}	1

may search for a sequence of these "Boolean differentials" (related to trun-

cated differentials [97, 99]) such that the MixColumns$\star$ transitions happen with a large probability. In every input round, the attacker obtains some degrees of freedom from the message input. A message input of all passive bytes gives the attacker 2^{μ} possibilities to choose the input. If there are d active bytes in the input, then the attacker has almost $2^{8d+\mu}$ degrees of freedom. Since the message does not affect the entire state after one round, the attacker can exercise some control using the message input for up to three rounds. For instance, he can choose an active byte in (almost) 2^{16} different ways until two particular bytes three rounds later have the desired differences. Other bytes are also affected by this variation of the active message byte, but not all bytes in the state are affected. Note that if the number of active bytes in the state is reduced slowly in order to form a collision, then (1) the MixColumns$\star$ transition cost may be relatively low, and (2) the degrees of freedom from the message input is relatively high (because the attack spans several rounds, and hence in total, many degrees of freedom are introduced via the message blocks).

If there exists a sequence of Boolean differentials such that the cost in terms of MixColumns$\star$ transitions can be handled by the degrees of freedom from the message input, and this characteristic leads to a zero-difference state, then an attack can be launched.

This is exactly what Peyrin did in his attack [154] on Grindahl-256. He found a sequence of Boolean differentials starting from a state where all bytes are active, ending in a colliding state, such that the total cost of the attack is about 2^{112}. A birthday collision attack on Grindahl-256 has complexity about 2^{128}. Hence, Grindahl-256 was broken.

Summary

Grindahl turned out not to have the collision resistance expected by the designers. The reason seems to have been a combination of the state being too small, and diffusion happening too slowly. Hence, it may be that variants of Grindahl with, say, 8 rows and 9 columns are secure. Alternatively, it seems that with four rows the number of columns needs to be at least 19. A third alternative is to apply the permutation P more than once for each message block. A speed/security trade-off is possible with a larger message block of, say, d columns, and calling the permutation r times for each.

3.4.4 Dakota

Dakota is a hash function with a security proof. It was designed by Damgård, Knudsen, and this author and presented at ACNS 2008 [49]. It was

inspired by a digital signature scheme of Goldwasser, Micali, and Rivest [76].

Security proofs for hash functions usually come in the form of an efficient reduction from an algorithm that finds a collision, to an algorithm that solves some problem which is generally believed to be difficult. Hence, the hash function is collision resistant based on the assumption of the intractability of the underlying problem. Most often, nothing is said about other properties of the hash function such as preimage resistance and randomness. In fact, it is often easy to show non-random behaviour of a hash function that is provably collision resistant (e.g., [183]). The security proof of DAKOTA concerns itself only with collision resistance, and hence it may have undesirable non-random properties. We propose to apply an output transformation in order to fix these issues.

Cryptographic hash functions with a security proof are often very inefficient compared to, e.g., the MD4 family (Section 3.4.2). The reason is that the underlying hard problem consists in doing some operation in a large group, finite field, or other mathematical structure, and doing arithmetic in such a large mathematical structure is inherently inefficient compared to logical operations such as XORs etc. DAKOTA is the result of an attempt to combine the world of large integer arithmetic with the world of symmetric cryptography, where logical and other simple operations are used. This way, we achieved a hash function which comes with a security proof, that does not quite cover the whole hash function, but on the other hand the hash function is much more efficient than existing hash functions based on large integer arithmetic.

Introducing DAKOTA

The Goldwasser-Micali-Rivest (GMR) signature scheme can be seen as a hash function that operates as follows. Denote by $QR(N)$ the set of squares (quadratic residues) modulo N, where N is a secure RSA modulus, in other words the product of two distinct, large secret primes p and q. Here, we furthermore require that $p \equiv q \equiv 3 \bmod 4$, which means that N is a *Blum integer* [131, Definition 2.156], and $-1 \notin QR(N)$. Split the message M into its individual bits b_i. Define two random squares a_0, a_1 modulo N. Define the compression function $f : QR(N) \times \{0,1\} \to QR(N)$ as

$$f(h, b) = a_b h^2 \bmod N$$

(the initial value of the hash function is a random square). A collision for f consists of two pairs (h, b) and $(\tilde{h}, \tilde{b})$, such that

$$a_b h^2 \equiv a_{\tilde{b}} \tilde{h}^2 \pmod{N} \iff a_b / a_{\tilde{b}} \equiv (\tilde{h}/h)^2 \pmod{N}.$$

Hence, $\tilde{h}/h$ is a square root of $a_b/a_{\tilde{b}}$. We distinguish the two cases $b = \tilde{b}$ and $b \neq \tilde{b}$.
If $\underline{b = \tilde{b}}$, then $\tilde{h}/h$ is a square root of 1. There are four square roots of 1 modulo N: $-1, 1, -d, d$. Since h and $\tilde{h}$ are squares, the quotient $\tilde{h}/h$, which is itself a square, cannot be -1. If it is 1, then $h = \tilde{h}$, and there is no collision. Hence, if $b = \tilde{b}$ then a non-trivial square root d or $-d$ of 1 has been found, which leads to the ability to factor N [131, Fact 3.46] by computing the greatest common divisor (gcd) of $d + 1$ and N.
If $\underline{b \neq \tilde{b}}$, then a square root of $a_b/a_{\tilde{b}}$ has been found. This is as difficult as factoring N, since a_0 and a_1 were randomly chosen.

Hence, finding a collision for this hash function is as difficult as factoring N. However, the hash function is extremely inefficient, since it takes a modular multiplication and a modular squaring to digest each bit of the message.

By choosing 2^μ random squares $a_0, \ldots, a_{2^\mu-1}$ instead of only two, efficiency is improved by a factor μ, and the security proof still holds. However, maintaining 2^μ random squares becomes impractical even for small values of μ such as 16 ($2^{16} = 65536$). Therefore, it would be convenient to have a function $g : \{0,1\}^\mu \to QR(N)$ that outputs a square given a μ-bit input string. However, it seems that such a function would have to square a known value, meaning that the security proof would break down.

If, on the other hand, one would apply a one-way function e to the μ-bit input before squaring it, yielding the compression function

$$f(h, m) = (e(m))^2 h^2 \bmod N = (e(m)h)^2 \bmod N, \tag{3.5}$$

then the security proof might still hold, since there would be no way of obtaining m from $e(m)$. Like f, e has to be one-way, and also collision resistant, so are we not just trapped in a vicious circle? No, not necessarily, since e does not need to compress, and therefore it may be injective meaning no collision *exists* in e.

We need to give a formal proof of security for this construction. For the proof, we need the to make the following assumption.

Assumption 3.1. It is computationally infeasible to find $x, \tilde{x}, z$, with $x \neq \tilde{x}$, such that

$$\frac{e(x)}{e(\tilde{x})} = \pm z^2 \bmod N.$$

Moreover, we assume that the initial value h_0 of the hash function is a square, for which no square root is publicly known. The hash function, which we denote by H, iterates f as defined in (3.5) in Merkle-Damgård mode. With these assumptions in place, the following theorem can be proved.

Theorem 3.2. *The hash function H as defined above is collision resistant under Assumption 3.1.*

Proof. Let $\mathcal{A}$ be an algorithm that finds collisions for H given an arbitrary initial value h_0, and an arbitrary modulus N, with probability ϵ in time T. We shall describe how $\mathcal{A}$ can be used to break Assumption 3.1 with probability at least $\epsilon/2$, in time equivalent to just a few computations more than T.

We are given a secure RSA modulus N. We choose an element s of $\mathbb{Z}_N$ at random, and set $h_0 = s^2 \bmod N$. We give h_0 and N to $\mathcal{A}$, and get the message pair $M, \tilde{M}$ such that $M \neq \tilde{M}$ and $H(M) = H(\tilde{M})$ in return.

Let $M = m_1 \| \cdots \| m_t$ and $\tilde{M} = \tilde{m}_1 \| \cdots \tilde{m}_{\tilde{t}}$, where each block is μ bits in length (padding is ignored here for simplicity, but makes no difference to the validity of the proof). Let h_i and $\tilde{h}_i$ be the intermediate hash values when processing M and $\tilde{M}$, respectively. We have $h_t = \tilde{h}_{\tilde{t}}$, and therefore $(e(m_t)h_{t-1})^2 \equiv (e(\tilde{m}_{\tilde{t}})\tilde{h}_{\tilde{t}-1})^2 \pmod N$. If $e(m_t)h_{t-1} \neq \pm e(\tilde{m}_{\tilde{t}})\tilde{h}_{\tilde{t}-1}$, then a factor of N can be found by computing, e.g., $\gcd(e(m_t)h_{t-1} - e(\tilde{m}_{\tilde{t}})\tilde{h}_{\tilde{t}-1}, N)$ [131, Fact 3.18], and by knowing the factorisation of N we can break Assumption 3.1. Hence, we may assume that $e(m_t)h_{t-1} = \pm e(\tilde{m}_{\tilde{t}})\tilde{h}_{\tilde{t}-1}$. This means that $e(m_t)/e(\tilde{m}_{\tilde{t}}) = \pm\tilde{h}_{\tilde{t}-1}/h_{t-1}$, where the quotient on the right hand side is a square. Assume first that $\underline{t = \tilde{t}}$. Then we know a square root of $\tilde{h}_{\tilde{t}-1}/h_{t-1}$, and hence, we have either broken Assumption 3.1, or $m_t = \tilde{m}_{\tilde{t}}$ and $h_{t-1} = \tilde{h}_{\tilde{t}-1}$ (since -1 is not a square modulo N, it cannot be that $h_{t-1} = -\tilde{h}_{\tilde{t}-1}$). In the latter case, the argument can be repeated. Note that this reasoning also applies when $t = \tilde{t} = 1$, in which case one has found a pair $(m, \tilde{m})$ such that $e(m) = \pm e(\tilde{m})$, leading to a contradiction with Assumption 3.1 with $z = 1$.

Assume now that $\underline{t \neq \tilde{t}}$, or more specifically and without loss of generality, that $t < \tilde{t}$. If $t > 1$, then the same argument as above for $t = \tilde{t}$ applies. If $t = 1$, then we have $e(m_1)/e(\tilde{m}_{\tilde{t}}) = \pm\tilde{h}_{\tilde{t}-1}/h_0$. We may assume that $m_1 = \tilde{m}_{\tilde{t}}$, since otherwise we have broken Assumption 3.1. Hence, we have $h_0 = \tilde{h}_{\tilde{t}-1}$, which means we can find a square root of h_0. With probability $1/2$, this square root is not equal to $\pm s$, which means that it can be used to factor N, and thereby break Assumption 3.1. Hence, given a collision from $\mathcal{A}$ we can break Assumption 3.1 with probability $1/2$, using just a few compression function evaluations and (possibly) a gcd computation. □

We remark that the factorisation of N must be unknown to the whole world for this hash function to be useful in practice. This may not seem possible, but there are in fact efficient and secure techniques [25] for constructing an RSA modulus in a distributed fashion in such a way that even the parties

taking part in the construction do not subsequently know the factorisation of N.

What requirements do Assumption 3.1 induce on e?

Requirements on e

It is clear that finding a collision for e leads to the ability to break Assumption 3.1 with $z = 1$.

Also, being able to invert e leads to the ability to break the assumption: choose $\tilde{x}, z$ arbitrarily, and compute $x = e^{-1}(e(\tilde{x})z^2)$. This, however, assumes that e is surjective. We may design e to expand by a large factor (i.e., choose $\mu \ll \log_2 N$), such that even finding $\tilde{x}, z$ such that $\pm e(\tilde{x})z^2$ is in the image of e is infeasible. The more that e expands, however, the less we gain in terms of efficiency compared to the GMR hash function.

Being collision resistant and one-way are not sufficient conditions, however. As an example, consider $e(x) = x^2 \bmod N$, which is assumed to be collision resistant and one-way when the input is restricted to values below $N/2$. Even ignoring the fact that a zero of this function is trivially found (which leads to a contradiction of the assumption with $z = 0$), this definition of e is bad because it has a simple description in terms of arithmetic modulo N. The equation that must be solved to break the assumption now becomes

$$\frac{x^2}{\tilde{x}^2} = \pm z^2 \bmod N,$$

which is satisfied with, e.g., $x = \tilde{x}z$ for almost any $\tilde{x}, z$ ($z \neq 1$).

Instead it may be better to define e based on principles from symmetric cryptography. We now give some proposals.

Proposals for e

The first proposal for e tries to exploit the good properties of squaring modulo a secure RSA modulus, while avoiding the negative side effect of the function having a simple description in terms of arithmetic modulo N. It applies a squaring modulo a modulus that is a few bits shorter than N, followed by the application of a permutation based on the AES encryption function [145].

To be precise, e is defined as follows. Let the input x have length μ bits. Let N' be a secure RSA modulus such that $2^\mu < N'/2$, and such that the number of bits in N' is a multiple of 128. N should be a few bits longer than N'. Let e do the following, where E_K is the AES encryption function with 128-bit key K, and κ_1 and κ_2 are two fixed, distinct 128-bit keys.

1: Compute $u \leftarrow x^2 \bmod N'$ and let $u = u_1 \| \cdots \| u_\ell$ ($|u_i| = 128$)

2: $v_0 \leftarrow 0$
3: **for** $i = 1$ to ℓ **do**
4: $\quad v_i \leftarrow E_{\kappa_1}(v_{i-1} \oplus u_i)$
5: **end for**
6: $w_0 \leftarrow 0$
7: **for** $i = 1$ to ℓ **do**
8: $\quad w_i \leftarrow E_{\kappa_2}(w_{i-1} \oplus v_{\ell-i+1})$
9: **end for**
10: **return** $w = w_1 \| \cdots \| w_\ell$

The order of the blocks v_i is reversed for the second series of encryptions (line 8) in order to make every output bit depend on every input bit.

We claim that this definition of e provides a one-way and collision resistant function, since $x < N'/2$. Moreover, we believe that encrypting the output of the squaring twice will make it difficult to retain simple relations modulo N. The relative sizes of μ, N' and N can be different than stated here, but it is convenient if the size of N' matches a multiple of 128 bits. Moreover, the output of e should be less than N, but for efficiency reasons, not too much less.

The second proposal is the application of an AES-based permutation P in a mode that can be seen as Matyas-Meyer-Oseas (Construction 3.2), namely $e(x) = P(x) \oplus x$. In the following description, we assume a 1025-bit modulus, and $\mu = 1024$. The invertible function T maps 1024-bit strings to 8×8 matrices of 16-bit values.

1: $A \leftarrow T(x)$
2: **for** $i = 1$ to 4 **do**
3: $\quad$ Encrypt (using the AES) each row in A with its own key
4: $\quad$ Transpose A (i.e., $A \leftarrow A^{\mathrm{T}}$)
5: **end for**
6: **return** $x \oplus T^{-1}(A)$

The security proof [24] of the Matyas-Meyer-Oseas construction can be extended to this construction, assuming the permutation applied is ideal. This means that the construction can be considered one-way and collision resistant. Of course, we cannot prove ideality of the permutation, but we claim that the permutation provides sufficient mixing for the resulting function to be one-way, collision resistant, and to make it very difficult to break Assumption 3.1.

If a larger modulus, say a 2049-bit modulus, is required, then T may map 2048-bit strings to 16×16 matrices of bytes instead.

Output transformation

In most applications, it is recommended that the output of H is not used as the final hash, but is instead fed to an output transformation function $\Omega : QR(N) \rightarrow \{0,1\}^n$, where n is chosen such that the complexity of factoring N is expected to be no less than $2^{n/2}$. The intention is to obtain an output size corresponding to the security level of the hash function. In addition, Ω should obfuscate the algebraic structure of the output. Any algebraic structure could compromise the security of schemes in which the hash function is used, particularly schemes that are based on modular arithmetic such as signature schemes based on RSA [174, 181]. A third purpose of an output transformation may be to improve the preimage resistance of the hash function.

In order to provably extend the collision resistance of H to $\Omega \circ H$, Ω would itself have to be collision intractable. But in practice, this may not be necessary: even if it is easy to find collisions for Ω, these do not necessarily lead to collisions for $\Omega \circ H$. In fact, it may be sufficient that Ω mixes the bits of its input well such that all output bits depend on all input bits, and that it is balanced. Unless the hash function is primarily used for short messages, Ω does not have to be terribly fast, since it is only used once.

A concrete example of how the output transformation could work is as follows: Let G be a finite group of prime order and let $g_1, ..., g_u$ be chosen randomly in G. We choose the two integers t and u such that $t < \log_2 |G|$ and $tu \geq \log_2(N)$. Let $b = H(x) = b_1 \| \cdots \| b_u$ be the output from the main hash function, where the length of each b_i is t. Since $tu \geq \log_2(N)$, this may require that b_u be padded with zeros – a simple padding scheme is fine here, since all inputs to the output transformation will have the same length. We then define the intermediate output $\omega(b)$ as $g_1^{b_1} \cdots g_u^{b_u}$, that is, a single element from G.

It is well known [33] (and straightforward to show) that finding collisions for this mapping is as hard as solving the discrete logarithm problem (DLP) in G. There are well known constructions of such groups based on elliptic curves, where the DLP can be reasonably assumed to be hard, and where the representation of a group element is 200-400 bits long. Note also that by choosing t small, for instance ≤ 8, we may perform the exponentiations $g_i^{b_i}$ by table look-ups.

We recommend that a final bijective function based on symmetric cryptography is used on $\omega(b)$ to produce the final output $\Omega(b)$, in order to prevent the adversary from exploiting the algebraic properties of exponentiation in G. This function could be designed in a similar way as in the first proposal for e above.

Related hash functions

A number of hash functions claiming to obtain provable security have been proposed in the past. We already described the GMR hash function. Some other examples are now mentioned.

Discrete log hash. The discrete log hash , or the Chaum-van Heijst-Pfitzmann hash function [33] is defined as follows. Let p and $q = \frac{p-1}{2}$ be large, odd primes. Let α and β be randomly chosen primitive elements of $\mathbb{Z}_p$, such that $\log_\alpha(\beta)$ is hard to find. Define the compression function $f : \mathbb{Z}_q \times \mathbb{Z}_q \to \mathbb{Z}_p^*$ by

$$f(h, m) = \alpha^h \beta^m \bmod p.$$

It can be shown that a collision for f enables one to compute $\log_\alpha(\beta)$.

MASH. The MASH (for Modular Arithmetic Secure Hash) functions [87] are standardised in ISO/IEC 10118-4. The compression function of MASH-1 is defined as follows. Let N be an RSA modulus, and let the message block be expanded to m, where the 4 most significant bits of every byte are set to 1111 (except in the final (padding) block, where the bits 1010 are inserted). Let $a = \mathsf{f00\ldots00}$ (in hexadecimal), and let

$$f(h, m) = \big(((h \oplus m) \vee a)^2 \bmod N\big) \oplus h.$$

In MASH-2, the exponent 2 is replaced by $2^8 + 1$. (See also [131, Algorithm 9.56]).

The MASH functions fall somewhat outside the category of provably secure hash functions, since no security proof exists. The claimed security of both these hash functions is $N^{1/2}$ for preimages, and $N^{1/4}$ for collisions.

VSH. A recent example of a provably collision resistant hash function is VSH [34], which is roughly defined as follows. Let N be a public RSA modulus. Let $p_1, \ldots, p_k$ be public primes such that $\prod_{i=1}^{k} p_i < N$. Define the compression function $f : \mathbb{Z}_N^* \times \{0,1\}^k \to \mathbb{Z}_N^*$ by

$$f(h, m) = h^2 \prod_{i=1}^{k} p_i^{m_i} \bmod N,$$

where m_i is the ith bit of m. The security of this construction relies on the so-called Very Smooth Square Root Problem, which is connected to the difficulty of factoring. There is also a variant of VSH whose security is based on the discrete logarithm problem [117]. VSH is generally considered to be one of the most efficient hash functions with provable security.

Performance

DAKOTA may be compared to the basic version of VSH [34] as follows: in both hash functions, a multiplication and a squaring modulo N must be performed for each message block, plus an overhead which in the case of VSH is due to the computation of the product of small primes, and in our case is due to the evaluation of e. DAKOTA is likely to perform better for one important reason: the size of a message block in our case is up to $\log_2(N)$ bits, whereas in the case of (basic) VSH it is the largest number t such that the product of the first t primes is less than N (in [34], t is estimated to be approximately $\frac{\log N}{\log \log N}$). As an example, with N a 1024-bit modulus, the size of a message block in DAKOTA may be up to 1024 bits, and in VSH it would be 131 bits.

There may also be an important difference in efficiency between evaluating e and computing the product of up to t primes.

There are faster versions of VSH that use larger message blocks and precompute some products of the small primes. These versions require a larger modulus to be used, and reliably comparing fast VSH with DAKOTA requires implementations using similar optimisations, compilers, processors etc. According to measurements presented in [34], fast VSH with claimed security equivalent to factoring a 1024-bit modulus reaches speeds of around 840 cycles/byte.

DAKOTA has been implemented with both proposed definitions of e and random choices of the moduli N (1025 bits) and N' (1024 bits), initial value h_0, and AES keys κ_1 and κ_2. For the AES, we used our own implementation in C. For large integer arithmetic, the GMP (GNU Multiple Precision) arithmetic library [75] has been used (version 4.2.1). The implementation was compiled and run on two different platforms, see Table 3.3 for benchmarks (also included are benchmarks for SHA-256 [146], obtained from [46]). The 32-bit benchmarks refer to a test on a 2GHz Pentium M processor. The 64-bit benchmarks were obtained on an Intel Core 2 processor and are due to [10]. Further optimisations of the implementation are almost certainly possible.

We have not performed tests using larger modulus sizes, although in order to achieve collision resistance comparable to, e.g., SHA-256, a modulus size of about 3072 bits would be needed, according to estimates by NIST [150].

Summary

DAKOTA is a cryptographic hash function with a security proof based on a non-standard assumption, that involves a function e which is required to

Table 3.3: Speed comparison of DAKOTA with SHA-256 (see text for details).

Hash function	Approximate speed (cycles/byte)	
	32-bit	64-bit
SHA-256	21.5	21.5
DAKOTA (first proposal for e)	400	160
DAKOTA (second proposal for e)	345	160

have certain properties. Among these are that e should be collision resistant and one-way, but since e does not compress, there is no circular argument such as "in order to build a secure compression function we just need a secure compression function". On the other hand, it may be argued that if e satisfies the requirements induced by the assumption, then e might itself be useful as a compression function if its output is truncated. We cannot prove this claim to be right or wrong. Clearly, if e is modelled as a random oracle, then the assumption is satisfied, but then e would already be a good compression function (upon truncation). However, we argue that the requirements on e are strictly weaker than the usual requirements on a compression function.

3.4.5 ANACONDA

In this section, we describe the ANACONDA hash function. ANACONDA is an attempt at designing a hash function achieving the good diffusion properties of the 4×4 matrix structure used in the block ciphers Square [41] and Rijndael/AES [42, 145]. The challenge in this respect is to arrive at a larger state size, since the 128-bit state of these block ciphers is not large enough for hashing. There are, of course, several methods to increase the state size; one method is to increase the size of the matrix (this method was used in the `Grøstl` hash function, see Section 5.1), and another method is to increase the word size from 8 bits to, e.g., 64 bits. The latter method was followed in the design of ANACONDA.

Unfortunately, 64-bit (and also 32-bit) S-boxes are infeasible to use in practice. Therefore, increasing the word size necessitates other changes as well. The non-linear transformation was therefore changed from operating on a word, to operating on a column, but in a bitslice fashion (as in Serpent [3]). Apart from introducing non-linearity, this also introduces diffusion within each column, and offers better protection against some side-channel attacks [111] than a table-based S-box.

On the other hand, a bitslice S-box does not introduce diffusion within a word. Hence, this must be taken care of by other means. The MixColumns

transformation known from Rijndael ensures maximal diffusion *among* words, but it, too, does not introduce much diffusion within a word. Therefore, it was decided to abolish the principles underlying the MixColumns transformation, and instead focus on an efficient linear transformation, that provides a large amount of diffusion within words, and also provides (sub-optimal) diffusion among the words in each row. The diffusion within words is obtained via a primitive known from the SHA-2 hash functions [146].

The end result is a hash function that bears resemblance with both Rijndael and Serpent at the same time. It is simple and (in the author's view) elegant.

The name Anaconda refers to a class of hash functions returning outputs of any size between 1 and 512 bits. The variants returning outputs of sizes between 1 and 256 bits all use the same method, only the initial value and the amount of truncation taking place in the end are different. This method is described in detail in the following. For the larger variants, another method is used; this method is described at the end of this section.

Anaconda was never submitted to a conference or a journal, but was published as a technical report [206]. The reason for including the design in this thesis is, that we believe it has some interesting properties, and the process of designing Anaconda brought with it some interesting observations. Among these are the observations on the Σ functions, described below, and the attacks described after the hash function specification. We believe it is also interesting to compare Anaconda with `Grøstl`, since both are, in a sense, methods of "enlarging" Rijndael (with very different outcomes).

Specification of Anaconda

The Anaconda hash function H takes messages of length up to about 2^{64} bits and returns a hash result of n bits, where n is any number between 1 and 512. We now describe how to produce hash results up to 256 bits.

Anaconda assumes a big-endian byte ordering. It applies a compression function $f : \{0,1\}^{512} \times \{0,1\}^{512} \rightarrow \{0,1\}^{512}$ in standard Merkle-Damgård mode. The output of the final application of the compression function is truncated to an n-bit value.

The compression function f operates with a 16-word state, that is seen as a 4×4 matrix of words (as in Rijndael). A state $A = a_0 \| \cdots \| a_{15}$ is seen as the matrix

$$A = \begin{bmatrix} a_0 & a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 & a_7 \\ a_8 & a_9 & a_{10} & a_{11} \\ a_{12} & a_{13} & a_{14} & a_{15} \end{bmatrix}.$$

Each word is 64 bits in size. Let $\mathcal{W} = \{0,1\}^{64}$. The compression function takes a 512-bit message block and a 512-bit chaining value, forms a 1024-bit state viewed as the matrix above, and applies a number of rounds to the state. The final state is then truncated to 512 bits. The round function is now described.

Round function. The round function $p : \mathcal{W}^{16} \to \mathcal{W}^{16}$ is a permutation. It applies two different transformations: the linear transformation $\mathsf{lt} : \mathcal{W}^4 \to \mathcal{W}^4$, and the non-linear transformation $\mathsf{nt} : \mathcal{W}^4 \to \mathcal{W}^4$. We shall come back to how these transformations are defined in a moment. The round function operates on a state $A = a_0 \| \cdots \| a_{15}$ as follows.

$$
\begin{array}{rcl}
a_{15} & \leftarrow & a_{15} \oplus 1 \\
(a_0, a_1, a_2, a_3) & \leftarrow & \mathsf{lt}(a_0, a_1, a_2, a_3) \\
(a_7, a_4, a_5, a_6) & \leftarrow & \mathsf{lt}(a_7, a_4, a_5, a_6) \\
(a_{10}, a_{11}, a_8, a_9) & \leftarrow & \mathsf{lt}(a_{10}, a_{11}, a_8, a_9) \\
(a_{13}, a_{14}, a_{15}, a_{12}) & \leftarrow & \mathsf{lt}(a_{13}, a_{14}, a_{15}, a_{12}) \\
(a_0, a_4, a_8, a_{12}) & \leftarrow & \mathsf{nt}(a_0, a_4, a_8, a_{12}) \\
(a_1, a_5, a_9, a_{13}) & \leftarrow & \mathsf{nt}(a_1, a_5, a_9, a_{13}) \\
(a_2, a_6, a_{10}, a_{14}) & \leftarrow & \mathsf{nt}(a_2, a_6, a_{10}, a_{14}) \\
(a_3, a_7, a_{11}, a_{15}) & \leftarrow & \mathsf{nt}(a_3, a_7, a_{11}, a_{15})
\end{array}
$$

See also Figure 3.9. In words, first a linear layer is applied to each row, and then a non-linear layer is applied to each column. The linear layer provides diffusion both on the bit-level and on the word-level (row-wise). The non-linear layer is a 4-bit S-box in bitslice mode, which provides diffusion on the word-level (column-wise). Notice that the order of the input words to the lt function is shifted for each row, and that the least significant bit of a_{15} is flipped in the beginning. These measures are in order to introduce asymmetry.

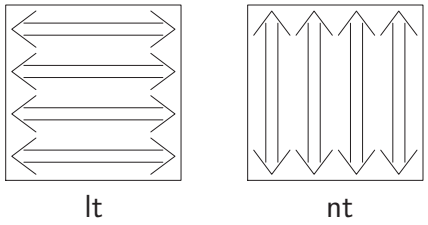


Figure 3.9: The effects of the transformations lt and nt.

Compression function. The input to the compression function f is the message block m and the chaining value h. From these, f forms the state $A = m\|h$. It then applies the round function p a number ℓ of times. ℓ is a security parameter, and hence its value may be chosen depending on the desired security level. We suggest $\ell = 16$. Since p is a permutation, no collisions are formed (yet).

After the ℓ applications of p, the compression function returns $\mathsf{trunc}_{512}(A)$. Hence, this final truncation is the only operation that introduces collisions. To be more precise, the compression function f is defined as follows:

$$f(h, m) = \mathsf{trunc}_{512}(p^{\ell}(m\|h)).$$

Linear transformation lt. In the definition of lt, four transformations operating on a single word each are applied. Let these be Σ_i, $0 \leq i < 4$. They are defined as follows.

$$\begin{array}{lcl} \Sigma_0(a) & \leftarrow & a^{\lll 1} \oplus a^{\lll 20} \oplus a^{\lll 24} \\ \Sigma_1(a) & \leftarrow & a^{\lll 13} \oplus a^{\lll 22} \oplus a^{\lll 60} \\ \Sigma_2(a) & \leftarrow & a^{\lll 30} \oplus a^{\lll 50} \oplus a^{\lll 63} \\ \Sigma_3(a) & \leftarrow & a^{\lll 24} \oplus a^{\lll 51} \oplus a^{\lll 54} \end{array}$$

Given the four input words (a, b, c, d), lt is defined as follows.

$$\begin{array}{lcl} a & \leftarrow & \Sigma_0(a) \\ b & \leftarrow & \Sigma_1(b) \\ c & \leftarrow & a \oplus b \oplus c \\ d & \leftarrow & a \oplus (b^{\gg 1}) \oplus d \\ c & \leftarrow & \Sigma_2(c) \\ d & \leftarrow & \Sigma_3(d) \\ a & \leftarrow & a \oplus c \oplus d \\ b & \leftarrow & b \oplus (c^{\gg 3}) \oplus d \end{array}$$

See Figure 3.10.

Non-linear transformation nt. As mentioned, the non-linear transformation is a 4-bit S-box in bitslice mode. The S-box S used for nt is defined as follows.

x :	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$S(x)$:	2	9	8	3	11	5	7	14	12	15	1	4	6	0	10	13

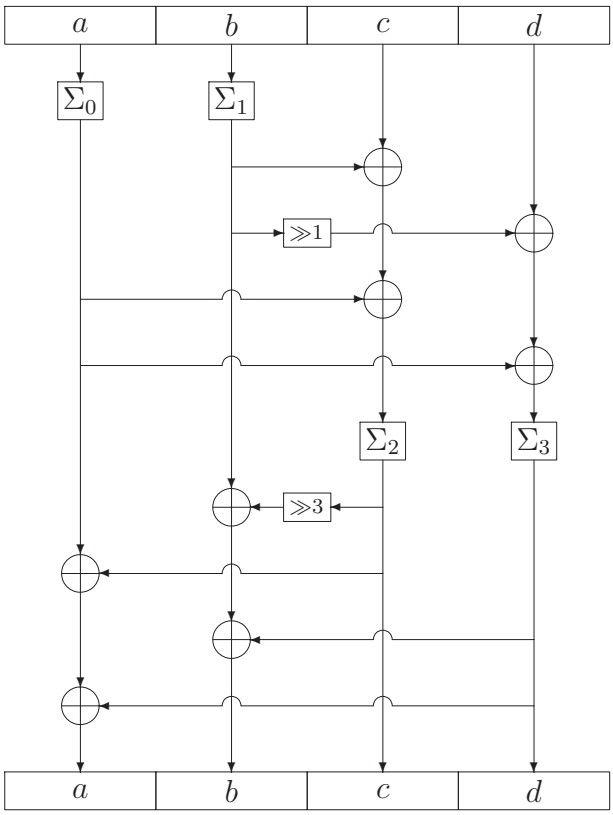


Figure 3.10: The linear transformation lt.

This S-box is, conceptually, applied as follows. Place the jth bit of the ith input word (counting from 0) in position (i, j) in a 4×64 matrix of bits. Apply the S-box to each of the 64 4-bit words defined by the columns of this matrix, where the bits in the top row are the most significant bits. Then, map the matrix back to four 64-bit words. In practice, the S-box application can be done via logical operations on the four 64-bit words. A method to do this is described later.

Padding. The padding function used is $\text{pad}^*_{512,64}$, see Definition 2.2 (page 19). This induces a maximum length of messages that can be hashed to $2^{64} - 1$ bits (approximately 2^{55} message blocks).

The hash construction. The hash function applies the compression function f in Merkle-Damgård mode, with the addition that the final output is truncated to n bits. The initial value is the 512-bit representation of the output size n. To be more precise, the message M is padded to $M^+ =$

$m_1\|m_2\|\cdots\|m_t$, and we set $h_0 = \langle n \rangle_{512}$. Let

$$h_i \leftarrow f(h_{i-1}, m_i) \quad \text{for } i = 1, \ldots, t.$$

Finally, let $H(M) = \mathsf{trunc}_n(h_t)$.

Some observations

An alternative representation of the compression function f, assuming a single 1024-bit input x and $P = p^\ell$, is

$$f(x) = P(x) \bmod 2^{512}.$$

(Here, $P(x)$ is seen as a 1024-bit integer). Collisions can easily be found for the compression function: choose y_1 and y_2 to be distinct 1024-bit values such that $y_1 \equiv y_2 \pmod{2^{512}}$, and compute $x_1 = P^{-1}(y_1)$ and $x_2 = P^{-1}(y_2)$. Then the pair (x_1, x_2) forms a collision for the compression function. However, if P is a good permutation, then an attacker will have no direct control over x_1 and x_2.

A meet-in-the-middle preimage attack (see Section 4.1.4) can be launched in time 2^{256} as follows. Let $y = H(M)$ be a target image. Compute $v_i = f(h_0\|a_i)$ for 2^{256} arbitrary 512-bit values of a_i, $0 < i \leq 2^{256}$. Compute $b_i\|w_i = P^{-1}(z_i\|y)$ for arbitrary $(1024 - n)$-bit values of z_i, $0 < i \leq 2^{256}$. Find a match (i, j) between v_i and w_j. Then $H(a_i\|b_j) = y$. We note that the complexity is at least as high as the complexity of a brute force preimage attack, since $n \leq 256$. The memory requirements are 2^{256}.

The invertibility of the compression function also leads to almost trivial pseudo-attacks. For instance, a pseudo-preimage attack: given target image $y = H(M)$, choose arbitrary z, and compute $m\|h_0^* = P^{-1}(z\|y)$. Then m is a preimage of y when h_0^* is used as the initial value of the hash function. We argue that h_0^* cannot be controlled (except by the method of the above preimage attack), and therefore we do not consider these pseudo-attacks a threat.

Design properties and considerations

This section describes some properties of the Anaconda design, and some considerations that were made in the development process.

The linear transformation. The linear transformation was to some extent inspired by the block cipher Serpent. The Σ transformations do not appear in Serpent, but are used in the SHA-2 hash functions.

The linear transformation has the following effect on the four input words a, b, c, d, where primed values are the new values of the input words:

$$\begin{aligned} a' &\leftarrow \Sigma_0(a) \oplus \Sigma_2(c \oplus \Sigma_0(a) \oplus \Sigma_1(b)) \oplus \Sigma_3(d \oplus \Sigma_0(a) \oplus (\Sigma_1(b))^{\gg 1}) \\ b' &\leftarrow \Sigma_1(b) \oplus (\Sigma_2(c \oplus \Sigma_0(a) \oplus \Sigma_1(b)))^{\gg 3} \oplus \Sigma_3(d \oplus \Sigma_0(a) \oplus (\Sigma_1(b))^{\gg 1}) \\ c' &\leftarrow \Sigma_2(c \oplus \Sigma_0(a) \oplus \Sigma_1(b)) \\ d' &\leftarrow \Sigma_3(d \oplus \Sigma_0(a) \oplus (\Sigma_1(b))^{\gg 1}). \end{aligned}$$

We discuss the Σ functions below. Some figures describing the diffusion taking place in lt follow.

Every bit in a affects at least 50 output bits; every bit in b affects at least 42 output bits; every bit in c affects at least 8 output bits; and every bit in d affects at least 9 output bits. Hence, diffusion is more effective in a and b than in c and d. Since the input ordering is changed in each row, the effectiveness of the diffusion is spread evenly among all columns.

Looking at the outputs, every output bit of a is affected by at least 34 input bits; every output bit of b is affected by at least 24 input bits; every output bit of c is affected by at least 21 input bits; and every output bit of d is affected by at least 18 input bits.

One many also look at the inverse of the linear transformation, lt^{-1}. This can be used to say something about how a low-weight state can be reached. To compute the inverse, one simply performs the steps described in the specification in the reverse order, applying the inverse Σ functions (see below). To reach a weight 1 output in a, the input must have weight 39. The same is the case for b, except in a single bit position where a weight 38 input suffices. In c, a weight 1 output can be reached by an input of weight at least 74 (60 out of 64 output bits require a weight 111 input), and in d, a weight 1 output can be reached by an input of weight at least 112.

The Σ functions. The purpose of the Σ functions is to cause diffusion within words. Σ functions are also used in the SHA-2 hash functions (see Section 3.4.2). We define the class of Σ functions as the functions

$$\Sigma(a) \leftarrow a^{\lll r_1} \oplus a^{\lll r_2} \oplus a^{\lll r_3},$$

where a is a 64-bit value, and $0 \leq r_1, r_2, r_3 < 64$. Each choice of rotation values (r_1, r_2, r_3) defines a member of the class. We now state and prove two theorems regarding Σ functions.

Theorem 3.3. *For any set of rotation values, Σ is a bijection, and $\Sigma^{64}(x) = x$.*

Proof. We first prove that $\Sigma^{64}(x) = x$. We have:

$$\begin{aligned}\Sigma^2(x) &= x^{\lll 2r_1} \oplus x^{\lll r_1+r_2} \oplus x^{\lll r_1+r_3} \oplus x^{\lll r_1+r_2} \oplus x^{\lll 2r_2} \oplus x^{\lll r_2+r_3} \oplus \\ &\quad x^{\lll r_1+r_3} \oplus x^{\lll r_2+r_3} \oplus x^{\lll 2r_3} \\ &= x^{\lll 2r_1} \oplus x^{\lll 2r_2} \oplus x^{\lll 2r_3}.\end{aligned}$$

Repeating this reasoning, we have $\Sigma^{2^k}(x) = x^{\lll 2^k r_1} \oplus x^{\lll 2^k r_2} \oplus x^{\lll 2^k r_3}$. Since $x^{\lll 64d} = x$ for all integers d, we get $\Sigma^{64}(x) = x \oplus x \oplus x = x$. This also shows that $\Sigma^{64}(x) = \Sigma(\Sigma^{63}(x)) = x \iff \Sigma^{63}(x) = \Sigma^{-1}(x)$, and therefore, Σ is a bijection. We note that the proof extends to the general case of 2^k-bit words. □

Theorem 3.4. *Exactly one or three of the rotation values r_1, r_2, r_3 are odd $\iff$ $\Sigma^i(x) \neq x$ for all x and all i, $1 \leq i < 64$.*

Proof. Since $\Sigma^{64}(x) = x$, we have that if $\Sigma^{32}(x) \neq x$, then $\Sigma^i(x) \neq x$ for all x and all $i < 64$ (since the only divisor of 64 that is not a divisor of 32, is 64). From the proof of Theorem 3.3, we know that $\Sigma^{32}(x) = x^{\lll 32r_1} \oplus x^{\lll 32r_2} \oplus x^{\lll 32r_3}$. If r_i is even, then $x^{\lll 32r_i} = x$, and otherwise, $x^{\lll 32r_i} = x^{\lll 32}$. Hence, if one or all three of the r_i are odd, then $\Sigma^{32}(x) = x^{\lll 32}$, and otherwise, $\Sigma^{32}(x) = x$. Again, the proof extends to the general case of 2^k-bit words. □

Obviously, r_1, r_2, r_3 should be distinct – otherwise, Σ is only a rotation and does not provide any diffusion. We consider a Σ function with distinct rotation values "good" if

- $\Sigma^i(x) \neq x$ for all x and all i, $1 \leq i < 64$.
- The inverse function Σ^{-1} causes good diffusion.

Let us investigate some requirements for the inverse function Σ^{-1} to cause good diffusion. From the proof of Theorem 3.3, we have that $\Sigma^{-1}(x) = \Sigma^{63}(x)$, and hence we see that the inverse is the sum (XOR) of all terms of the form $x^{\lll a}$, where $a = r_{i_0} + 2r_{i_1} + 4r_{i_2} + 8r_{i_3} + 16r_{i_4} + 32r_{i_5} \bmod 64$, and $i_j \in \{1, 2, 3\}$. Intuitively, the inverse causing good diffusion corresponds to this sum having many terms that don't cancel out: for inputs of Hamming weight 1, the output Hamming weight is equal to the number of terms in the sum that haven't been cancelled out by other terms. For inputs of Hamming weight 2 to have high output Hamming weight, we need that $x^{\lll a} \oplus x^{\lll a+i}$ contains many terms for all i, $1 \leq i < 64$. There always exist inputs to Σ^{-1} of Hamming weight 3 with output Hamming weight 1, because if x has Hamming weight 1, then $\Sigma(x)$ has Hamming weight 3.

If r_1, r_2, r_3 are all odd, then a is always odd, and hence the inverse is the sum of at most 32 terms. We may obtain better results by requiring that r_1 is odd and r_2 and r_3 are even.

In order to find optimal rotation values, we performed a search. We fixed $r_1 = 1$ and searched for distinct, even values of r_2 and r_3 that would yield high output Hamming weights for inputs to Σ^{-1} with Hamming weights 1 and 2. Of course, this search could have been done without the theory described above, since the search space has size at most 2^{18}. It turned out that 60 pairs (r_2, r_3) produced equally good results: a weight 1 input to Σ^{-1} causes at least a weight 37 output, and a weight 2 input causes at least a weight 26 output. Note that translating the rotation values does not change the mentioned properties. By translating a set of rotation values, we mean adding the same even constant to all three rotation values.

In terms of the properties that we tried to optimise for Anaconda, the Σ functions used in SHA-2 are not optimal. For instance, $\Sigma_0^{\{512\}}$ (see [146]) has $(35, 18)$ in place of $(37, 26)$ (recall that these are the minimum output Hamming weights of Σ^{-1} given input Hamming weights $(1, 2)$). We assume there are other good reasons why the Σ functions used in SHA-2 were chosen as they were.

After having found the 60 equally good pairs, we chose four of them more or less at random, but such that the two even rotation values were quite different. We then searched for translations of these four sets of rotation values such that when used inside lt, a weight 1 input to $\mathsf{lt} \circ \mathsf{lt}$ yielded an output with a large Hamming weight. The first solution, for which the output Hamming weight was maximal, was chosen.

The reason why we chose to measure the diffusion of lt by using two applications of the linear transformation is that for only a single application, all translations have the same effect: a weight 1 input yields at least a weight 8 output. For $\mathsf{lt} \circ \mathsf{lt}$, with the Σ functions chosen for Anaconda, a weight 1 input yields at least a weight 120 output.

The inverse Σ functions have the following descriptions, where we list each of the 37 rotation values.

Σ_0^{-1}:	3 4 5 6 11 14 16 17 18 19 20 23 24 25 26 27 28 29 30 33 34 35 36 38 40 41 44 45 50 51 54 55 56 58 60 61 63
Σ_1^{-1}:	0 1 3 4 6 11 14 16 17 18 20 21 22 24 26 27 28 29 30 31 32 33 34 35 38 39 40 41 42 45 49 51 53 54 60 61 62
Σ_2^{-1}:	0 1 3 4 5 8 12 13 14 18 19 20 21 22 23 25 26 27 28 30 31 32 34 36 37 39 40 42 43 48 50 53 55 57 58 61 62
Σ_3^{-1}:	2 3 4 5 6 7 9 10 11 13 14 16 21 22 23 24 26 27 28 29 30 31 35 38 39 40 44 46 48 49 50 52 55 59 60 61 62

The non-linear transformation. The S-box used in ANACONDA is a variant of the Serpent S-box numbered 0. The change as compared to the original Serpent S-box is based on the Serpent implementation by Osvik [153]: here, each output bit is not in its right place after the S-box application, and subsequent word moves are done implicitly. We would like to avoid this, and hence we have just taken the output bits in the order they come out of the S-box application. Furthermore, a logical negation was omitted in order to speed up the implementation. This change only corresponds to an affine transformation, and hence does not change the important properties of the S-box.

The properties of the S-box are described in [3]. As an example, flipping one input bit causes at least two output bits to flip.

We might add that an S-box in bitslice mode offers better protection against side-channel attacks such as timing analysis [111], than an S-box implemented via table look-ups.

Global diffusion. The good diffusion properties of both the linear transformation and the non-linear transformation cause global diffusion to be very effective. Full diffusion occurs after just three rounds, which means that every state bit affects every other state bit after at most three rounds.

After just two rounds, half the state bits affect all other state bits, and every state bit affects at least 956 out of the 1024 state bits. On average, every state bit affects 1003 state bits after two rounds. Hence, diffusion is almost complete after just two rounds.

State size. Since the chaining value is at least twice as large as the output, ANACONDA is a wide pipe construction, see [121] and Section 3.3.2. An important advantage of this construction is that it allows "slight failures" in the compression function. Consider, for instance, a collision attack on the compression function, which works for any given chaining input. Assume this attack has complexity $2^{c/4}$, where c is the size of the chaining value. Although being an indication that the compression function does not provide ideal security, this attack would not constitute a collision attack on the hash function, since $c \geq 2n$ and therefore $2^{c/4} \geq 2^{n/2}$.

Implementation issues

As mentioned, the S-box was chosen from the set of Serpent S-boxes. These were implemented by Osvik [153], who focused on their performance on Pentium processors. A C implementation of the S-box used in ANACONDA (based on Osvik's implementations) follows (`t` is a temporary variable).

```
#define S(x0,x1,x2,x3,t) do { \
    t   = x3; \
    x3 |= x0; \
    x0 ^= t;  \
    t  ^= x2; \
    t  =~ t;  \
    x3 ^= x1; \
    x1 &= x0; \
    x1 ^= t;  \
    x2 ^= x0; \
    x0 ^= x3; \
    t  |= x0; \
    x0 ^= x2; \
    x2 &= x1; \
    x3 ^= x2; \
    x2 ^= t;  \
    x1 ^= x2; \
  } while (0)
```

More efficient implementations are likely to exist, particularly on other processors than the Pentium. It is generally difficult to find optimal implementations of bitslice S-boxes.

Since no words are copied from one position in the state matrix to another, all computations can be done "in place", and therefore an implementation using 1024 bits of memory is possible.

An implementation of ANACONDA in C has been developed [204] and run on an Intel Core2 Duo (E4600) 64-bit processor. With 16 rounds, the implementation reaches a speed of around 23 cycles/byte. The compiler used was Intel's C compiler version 10.1 (build 20080112) for Linux. The `gcc` compiler does not achieve the same speed for reasons that are unclear at this point. We note that improvements to the implementation are almost certainly possible.

Security claims

We claim that the best collision attack on ANACONDA variants returning up to 256 bits has complexity around $2^{n/2}$, and the best preimage and second preimage attacks have complexity around 2^n. See also the discussion at the end of this section.

Cryptanalysis

Further work on Anaconda was postponed due to the SHA-3 competition. The author decided to take part in another design, `Grøstl`, and hence, an investigation of the feasibility of known attacks on Anaconda has yet to be made. However, we note here that in each round, the 4-bit S-box is applied 256 times, totalling 4096 S-box applications over the 16 rounds. This number is huge compared to, e.g., the number of S-box applications in the 10 rounds of Rijndael, which is 160, or the number of S-box applications in the 32 rounds of Serpent, which is 1024. Of course, 4-bit and 8-bit S-boxes cannot be compared directly, and the number of degrees of freedom that an attacker has in the input to Anaconda is higher than in the block ciphers. However, considering the large number of S-box applications combined with the good diffusion taking place in Anaconda, we believe that established attack methods such as differential attacks will not work on Anaconda.

Larger variants

One possible method of building a 512-bit hash function on the basis of the method described above is to use 128-bit words instead of 64-bit words. Although many processors provide SSE instructions on 128-bit vectors, these are not capable of performing rotations, and hence we estimate that there would be a rather large penalty in terms of efficiency. Instead, we chose to change the definition of the compression function to the following.

$$f(h, m) = \mathsf{trunc}_{512}(p^{2\ell}(m \| h)) \oplus h,$$

or, in the alternative representation,

$$f(x) = P^2(x) \oplus x \bmod 2^{512}.$$

The chaining input is fed forward to protect against the meet-in-the-middle preimage attack mentioned above, which would have complexity below 2^n if $n > 256$.

The security claims for the larger variants of Anaconda are the same as for the shorter variants, except for second preimage resistance, which we claim to be at a level of 2^{n-k} compression function evaluations for a first preimage of 2^k blocks.

With 32 rounds, the larger variants perform at around 45 cycles/byte in the environment described above.

Summary

Anaconda is a hash function built on some of the design principles underlying Rijndael, and also on some of those underlying Serpent. The compression function (for variants returning up to 256 bits) is a permutation followed by a truncation. The internal state size is at least twice the output size. Collisions can easily be found in the compression function (even assuming the permutation is ideal), and therefore the security proof of the Merkle-Damgård construction does not apply to Anaconda. However, it does not seem possible to extend the collision attack on the compression function to the full hash function, assuming that the permutation contains no weaknesses.

For the larger variants returning more than 256 bits, it seems harder to find collisions for the compression function. The compression function is not invertible, due to a feed-forward of the chaining input. This feed-forward is omitted in the shorter variants in order to avoid having to store a copy of the 512-bit chaining input.

Like RadioGatún and Grindahl, the Anaconda hash functions can be seen as instances of the sponge construction (Section 3.3.5). However, Anaconda takes larger messages blocks, but applies a stronger permutation than RadioGatún and Grindahl.

Anaconda variants returning at most 256 bits can be seen as random P-sponges, for which we claim that the capacity is 512 bits. This would mean that the hash functions have optimal collision, preimage, and second preimage resistance.

Variants returning more than 256 bits can be seen as random T-sponges, with claimed capacity 512 bits. This means that collision and preimage resistance are claimed to be optimal, but second preimage resistance degrades with the length of the first preimage; if the length of the first preimage is L message blocks, then second preimage resistance is at most $2^{512}/L$. Hence, for the longest variants, finding a second preimage has sub-optimal complexity (note that the maximum value of L is 2^{55}). However, we argue that a second preimage resistance of $2^{512-55} = 2^{457}$ is sufficient for a 512-bit hash function, in particular when considering that this complexity requires the first preimage to have a length of 2^{55} message blocks (equivalent to about 2.3 million terabytes).

Chapter 4

Hash function cryptanalysis

Hash function cryptanalysis has evolved rapidly in the last few years. Most attacks have been collision attacks, but more recently, also a number of preimage attacks have appeared. In this chapter, we describe collision and preimage attacks on MD2 and on the MDC-2 construction. We describe how some of the generic attacks from Section 2.2 can be applied to checksum-based hash functions. Finally, we show how to find collisions in many permutation-based hash functions of rate 1/2. We start off with a description of some general techniques and notions that are often used in hash function cryptanalysis.

4.1 Introduction

Some techniques are used often in cryptanalysis, and in particular in this chapter. We give an introduction to some of these here.

4.1.1 Searching and sorting

In many attacks, a list or a table is filled with data, that later must be sorted and/or searched. Therefore we would like to give a brief introduction to searching and sorting (more details can be found in [36], and the information in this section is to a large extent taken from there).

Sorting a number N of elements takes time about $N \log(N)$ using, for instance, QuickSort. When the N elements are sorted, a particular element can be found in time $\log(N)$. This type of sorting is called *comparison sorting*.

If the nature of the elements to be sorted is known beforehand, then in most cases sorting can be done more efficiently without comparisons. For instance, assume that we have a list of N distinct elements, such that we know that each element a_i is in the range $0 \leq a_i < 2N$. Assume also that

each element a_i has some auxiliary data b_i associated with it (for instance, this may be a message block and its corresponding hash value). We may sort these elements by using a *direct address table*: Prepare a table T of size $2N$, and place b_i in T at the position a_i (we write $T[a_i] \leftarrow b_i$). This takes linear time. Looking up an element a_i and its auxiliary data b_i takes constant time.

If the elements a_i come from a set of size much greater than N, then it is more economic in terms of memory usage to use a *hash table*; here, an element is not inserted directly into the address a_i, but instead a function h is applied to a_i first. This function is, somewhat unfortunately, called a hash function. It has some of the same properties as a cryptographic hash function: it maps elements from a large set to a smaller set, and collisions are bad. But a hash function of this type is by no means cryptographically secure; collisions can, in general, easily be found, but they should not occur too often in practice. h may have a range of size about N, and if a collision on the index to the hash table T occurs, then it can be resolved by using, e.g., a linked list. To recap, the pair (a_i, b_i) is inserted into T as $T[h(a_i)] \leftarrow b_i$. We shall sometimes, for simplicity, write this in the same way as for the direct address table above, i.e., ignoring the hash function h. The complexity of sorting and searching is the same for a hash table as for a direct address table (h is assumed to be extremely efficient). In some of the attacks in this chapter we shall use tables, and we shall not always be exact about which kind of tables we are using.

4.1.2 Meaningful messages

When carrying out a brute force attack, a birthday attack, or some other type of attack, one often tries a large number of arbitrary messages until the attack succeeds. In practice, it might be required that the final message (or message pair) somehow contains "meaningful" information, such that it can be used to forge or otherwise practically compromise a cryptographic system.

A method suggested by Yuval [224] accomplishes this. We assume that the attack is of a nature such that in the end, a single message block is found with some desired property. A large number, say 2^k, of message blocks must be tried before one expects to find one with the desired property. One may start off by generating a single variant m of the message block, containing some desired piece of information. Then, one can make many variants m_i by substituting a word by a synonym, replacing a comma by a full stop, adding whitespace, replacing, e.g., "\$100,000" by "\$110,000", etc. If two equivalent wordings (or the like) can be used in k different positions in the message, then 2^k more or less equivalent messages can be generated.

In cases where the attack imposes many restrictions on the final message

or message pair, it is often still possible to end up with a message that is somehow meaningful. For instance, the message may be a document written in some language such as PostScript, PDF, Microsoft Word, HTML, etc. These languages provide the ability to hide part of the information that is contained in a document, e.g, as a comment, as characters having the same colour as the background, etc. The "random" part of the final message may even be used constructively, for instance as part of a random number written in the document. Many examples where this or a similar technique is used have been seen recently [73, 116, 118, 122, 198]. These examples show that the utility of a particular attack in practice may be greater than expected.

4.1.3 Memoryless collision search

The standard birthday collision attack on the n-bit hash function H is carried out as follows. Choose $2^{n/2}$ messages M_i, $1 \leq i \leq 2^{n/2}$, and compute $y_i = H(M_i)$ for each i. A pair (i, j), $i \neq j$, is expected to exist such that $y_i = y_j$. To find this pair, one has to go through all (or most) of the hash values y_i. As mentioned above, the list of hash values y_i can be sorted in linear time, and once sorted, it is easy to check for a collision (also in linear time).

The above approach requires storing $2^{n/2}$ messages and their hashes. However, this large memory requirement can be eliminated, or at the very least reduced significantly, by the use of a cycle-finding technique similar to that of Floyd [67], or Brent [27].

The idea of a cycle-finding method is to initiate a sequence where each value depends on the previous value. Since the values are of a finite length, at some point the sequence will start to cycle. If P is the first point in the sequence that is computed twice, then the part of the sequence before P is called the *tail*, and the part after P (including P itself) is called the *cycle*. See Figure 4.1. Some method of detecting when a sequence enters the cycle is needed, and some method of finding the value before P, both on the tail and on the cycle, is also needed. In the case of collision search, the sequence will be determined by the hash function H.

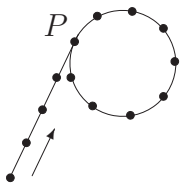


Figure 4.1: The cycle-finding method.

A cycle-finding algorithm was also used in Pollard's *rho* method for fac-

torisation [156], where two sequences of "random" values modulo the product are computed. In the context of collision search on a hash function, a technique using a single sequence, but requiring a small amount of storage, may be used (see also [131, Section 9.7.1]). (Hence, the term "memoryless" is a little misleading.)

We describe the memoryless collision search on the hash function H here. Choose an arbitrary message M of length at least n bits. Define a function $R : \{0,1\}^n \to \{0,1\}^n$ as

$$R(x) = H(M \oplus x).$$

R could be defined in a number of ways, but we chose a simple definition. A point $x \in \{0,1\}^n$ is a *distinguished point* if it satisfies some condition. We require here that a randomly chosen x is a distinguished point with probability about 2^{-d}. For instance, a distinguished point may be defined as a point that has '0' bits in all the last d positions.

The memoryless collision search proceeds as follows, where L is a list that is initially empty, and ip is some initial point. We note that this is a high-level description, ignoring some details.

1: $x_0 \leftarrow R(\text{ip})$ (add x_0 to L)
2: $i \leftarrow 0$
3: **repeat**
4: **repeat**
5: $i \leftarrow i + 1$
6: Compute $x_i \leftarrow R(x_{i-1})$
7: **until** x_i is a distinguished point
8: Add x_i to L
9: **until** $L[j] = L[j^*]$ for some (j, j^*), $j \neq j^*$, and $j, j^* > 0$
10: $y_0 \leftarrow L[j-1]$ and $y_0^* \leftarrow L[j^*-1]$
11: Compute (as above) the two sequences $y_0, y_1, \ldots$ and $y_0^*, y_1^*, \ldots$ until $y_u = y_v^*$ for some (u, v)
12: Since $y_u = y_v^*$, $R(y_{u-1}) = R(y_{v-1}^*) \iff H(M \oplus y_{u-1}) = H(M \oplus y_{v-1}^*)$
13: **return** $(M \oplus y_{u-1}, M \oplus y_{v-1}^*)$

A few things can go wrong in this algorithm, such as the algorithm entering a cycle containing no distinguished points. In this case, the algorithm as described here will go into an infinite loop, but this can be avoided by a few additional steps.

Since the probability that a randomly chosen point is a distinguished one is about 2^{-d}, the length of the inner loop (lines 4–7) is expected to be about 2^d. The length of the outer loop (lines 3–9), and thereby the expected number of distinguished points that need to be stored, is about $2^{n/2-d}$, since we expect the sequence to start cycling after about $2^{n/2}$ hash

values have been computed. Hence, d determines the memory requirements. On the other hand, line 11 also depends on d, and about 2^d points y_i and y_i^* must be computed and stored. However, distinguished points (with relaxed requirements) may be used again for this part, for instance with probability $2^{-d/2}$. This method can be repeated, so that the memory requirements are always about $2^{n/2-d}$. The expected time required is greater than for the usual memory-expensive variant described in the beginning of this section, but only by a small factor.

As an example, if $d = n/4$, memory requirements can be as low as $2^{n/4}$, and the expected time required is around $2^{n/2} + 2^{n/4}$. Another trade-off is $d = 3n/8$, with memory requirements around $2^{n/8}$ and expected time complexity around $2^{n/2} + 2^{3n/8}$.

This collision search can be parallelised as shown by van Oorschot and Wiener [208, 209]. It can also be generalised to r-collisions; when a distinguished point P is reached, it is stored, and if P has been seen before, a new sequence is initiated with a new initial point. Once a distinguished point has been seen in r different sequences, there is a chance that we have found an r-collision – but only a slight chance, since it is more likely that, say, three sequences meet at two different points before the distinguished point. We call this a *false alarm*. It must be checked whether a possible multi-collision is a false alarm, and if so, the search continues. The trade-off between time and memory using this multi-collision search method is not quite as good as above due to false alarms, but the first time a true r-collision occurs, it is found by this method.

4.1.4 Meet-in-the-middle attack

If the compression function f of a Merkle-Damgård hash function H is invertible, then a so-called *meet-in-the-middle attack* can be applied on H to produce a preimage. The method is the following, where we produce a preimage of y (we ignore padding of the message, but this can easily be accounted for).

1: **for** $i = 1$ to $2^{n/2}$ **do**
2: Choose an arbitrary message block m_2^i, and compute h_1^i such that $f(h_1^i, m_2^i) = y$.
3: Store m_2^i in the table T, indexed by h_1^i – in other words, let $T[h_1^i] \leftarrow m_2^i$.
4: **end for**
5: **repeat**
6: Choose an arbitrary message block m_1.
7: Compute $h_1 = f(h_0, m_1)$.

8: **until** $h_1 \in T$
9: **return** $m_1 \| T[h_1]$

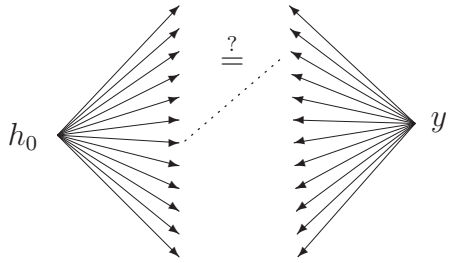


Figure 4.2: The meet-in-the-middle attack.

Since $f(h_0, m_1) = h_1$ and $f(h_1, T[h_1]) = y$, the message $m_1 \| T[h_1]$ is a preimage of y. The time required is about $2^{n/2+1}$: the first loop (lines 1–4) is executed $2^{n/2}$ times, and we assume that inverting f can be done in the same amount of time as computing it in the usual, forward direction. The second loop (lines 5–8) is also expected to take time $2^{n/2}$, since we are trying to match an n-bit value with any one of $2^{n/2}$ n-bit values.

The attack can be generalised: if it takes time $T = 2^\tau$ to invert f, then we may perform the first loop $2^{(n-\tau)/2}$ times, which would take time $2^{(n+\tau)/2}$. This is also the expected time required for the second loop. Note that τ is always between 0 and n.

4.1.5 Wagner's generalised birthday attack

Wagner's generalised birthday attack [213] is a method of finding collisions in functions that can be written as the XOR of a number of sub-functions, that each depend on its own input. For instance, a function f of two inputs x and y may be defined as $f(x, y) = f_1(x) \oplus f_2(y)$, where f_1 and f_2 are arbitrary functions. Functions f of this type allow a particularly efficient collision attack.

Assume that f, f_1, and f_2 are n-bit functions. Compute $f_1(x)$ for $2^{n/3}$ different values of x, and place the outputs in the list L_1. Construct in the same way the list L_2, but using different inputs. Construct the two lists L_3 and L_4 also in the same way, but using the function f_2 instead of f_1. Find all pairs of elements from L_1 and L_2, that agree on the last $n/3$ bits (this can be done in linear time). With $2^{2n/3}$ pairs in total, we expect $2^{n/3}$ pairs to match on $n/3$ bits. Store the XOR of these in the list U_1. Note that the last $n/3$ bits of all elements in U_1 will be zeros. Likewise, find matching pairs in the two lists L_3 and L_4, and store the XOR of these pairs in the list U_2.

With $2^{n/3}$ elements in U_1 and U_2, $2^{2n/3}$ pairs can be formed; one of these is expected to match on the remaining $2n/3$ bits.

When there is a match, we have found a quadruple with the property that $f_1(x) \oplus f_1(x^*) = f_2(y) \oplus f_2(y^*)$, meaning that $f(x, y) = f(x^*, y^*)$. An algorithmic version of the attack follows.

1: **for** $i = 1$ to $2^{n/3}$ **do**
2: Choose arbitrary x_i, compute $y_i = f_1(x_i)$, and do $L_1[y_i] \leftarrow x_i$
3: Choose arbitrary x_i^*, compute $y_i^* = f_1(x_i^*)$, and do $L_2[y_i^*] \leftarrow x_i^*$
4: Choose arbitrary x_i^+, compute $y_i^+ = f_2(x_i^+)$, and do $L_3[y_i^+] \leftarrow x_i^+$
5: Choose arbitrary $x_i^\#$, compute $y_i^\# = f_2(x_i^\#)$, and do $L_4[y_i^\#] \leftarrow x_i^\#$
6: **end for**
7: Find all pairs in L_1 and L_2 that match in the last $n/3$ bits of the output of f_1. For each pair $((x, y), (x^*, y^*))$, do $U_1[y \oplus y^*] \leftarrow (x, x^*)$.
8: Find all pairs in L_3 and L_4 that match in the last $n/3$ bits of the output of f_2. For each pair $((x, y), (x^*, y^*))$, do $U_2[y \oplus y^*] \leftarrow (x, x^*)$.
9: Find a collision (y, y^*) on the index to U_1 and U_2, i.e., an index for which there is an entry in both tables.
10: **return** $U_1[y], U_2[y^*]$.

Since f_1 and f_2 are evaluated $2^{n/3+1}$ times each, and some $2^{n/3+1}$ comparisons are needed, the total running time of the algorithm is about $2^{n/3+1}$ in terms of compression function equivalents. In the following, we shall simplify this complexity estimate to $2^{n/3}$. The memory requirements are the same.

The algorithm can be generalised to functions that can be written as the XOR of a number any power of two of sub-functions with distinct inputs. If there are 2^k sub-functions, then the complexity of the attack is about $k2^{n/(k+2)}$. The method also generalises to other operations than XOR, such as addition modulo 2^n.

4.2 Cryptanalysis of MD2

In this section we describe a number of attacks on the hash function MD2. See Section 3.4.1 for a description of the hash function.

Although, as we describe in the following, there is a number of successful cryptanalytic results on MD2, it is still used in practice and is part of several (de facto) standards, see e.g., [137]. The first cryptanalytic result on MD2 was a collision attack [179] (published in 1997 by Rogier and Chauvaud) on a reduced version of MD2 where the checksum block was omitted. The first attack against the full MD2 hash function was a preimage attack (of complexity around 2^{104}) published by Muller [137] in 2004. This attack was improved (to a complexity of about 2^{97}) by Knudsen and Mathiassen in [101],

where they also generalised and found further use of the collision attack by Rogier and Chauvaud.

In this section we describe some improvements and extensions of the mentioned attacks. Specifically, we describe the first collision attack on the MD2 hash function having complexity below the birthday level, namely around $2^{61.4}$. We also describe a major improvement to preimage attacks on MD2, having complexity around $2^{72.6}$. This preimage attack is, at the same time, a second preimage attack. Note that a birthday collision attack on MD2 has complexity around 2^{64}, and a brute force (second) preimage attack has complexity around 2^{128}. The results presented in this section are joint work with Lars R. Knudsen, John Erik Mathiassen, and Frédéric Muller [102]. First, we note some observations on the MD2 compression function.

4.2.1 Observations on the compression function

First, we note that the MD2 compression function can be expressed as in Figure 4.3. This view of the compression function is instructive when studying the attacks presented in this section. We shall often refer to the three rectangular structures as rectangles A, B, and C, since row i of, e.g., A is exactly A_i (as introduced in Section 3.4.1). Note that since there are 832 bytes in total

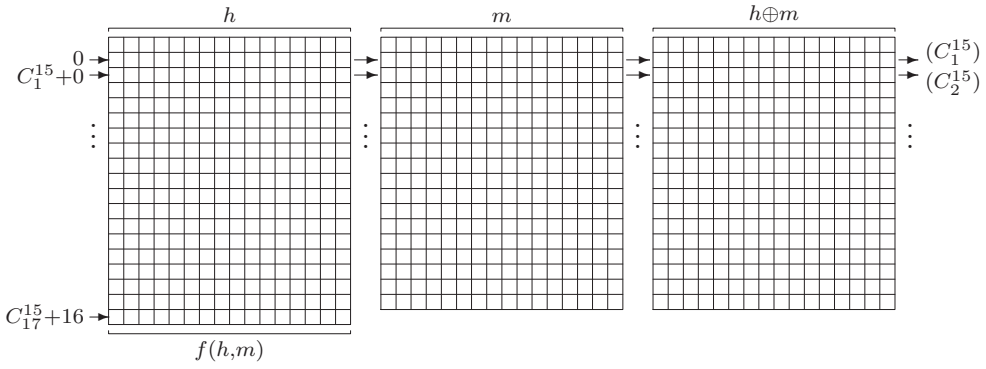


Figure 4.3: The MD2 compression function

in the three rectangles A, B, and C, in our complexity estimates we shall assume that computing one byte corresponds to 1/832 compression function evaluations. This will also be used as an estimate for a simple operation such as an XOR.

Since X_i^j is computed as the XOR of X_{i-1}^j and a bijective function of X_i^{j-1} (for $i > 0$ and $j > 0$), a cryptanalyst may instead compute either of

the three values from the two others in one of the following ways:

$$\begin{array}{rcll} X_i^j & = & X_{i-1}^j \oplus S(X_i^{j-1}) & (\text{“folding right”}) \\ X_{i-1}^j & = & X_i^j \oplus S(X_i^{j-1}) & (\text{“folding up”}) \\ X_i^{j-1} & = & S^{-1}(X_i^j \oplus X_{i-1}^j) & (\text{“folding left”}) \end{array} \tag{4.1}$$

For $i = 0$, X_i^j is not computed but is rather part of the input. $X_1^0 = A_1^0$ is computed from $X_0^0 = A_0^0 = h^0$ only.
For $j = 0$ a similar relationship exists, i.e., X_i^0 $(= A_i^0)$ is computed from X_{i-1}^0 and X_{i-1}^{47} $(= C_{i-1}^{15})$.

Comparing with Figure 4.3, this means that any “triangle” of the form

is completely determined by two of the squares. As an example, if the output of the compression function, which corresponds to the last row of A, is known, then by “folding up” (see (4.1)) one may compute the entire lower right triangle as indicated in Figure 4.4. In the following, we shall generally

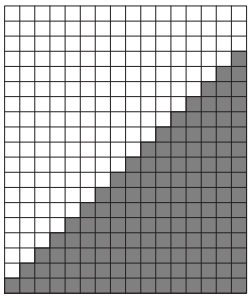


Figure 4.4: The shaded values are the values of the rectangle A that may be computed if the output of the compression function, corresponding to the last row of A, is known.

indicate known values as shaded squares of the rectangles A, B, and C.

4.2.2 The collision attack

Here we describe a collision attack on the MD2 hash function. The attack is based on a collision attack on the compression function, where the chaining input is arbitrary.

A collision attack on the compression function

The collision attack on the MD2 compression function described here allows the attacker to choose the chaining input h freely. The attack is largely inspired by the Scenario 1 preimage attack described in [137].

Assume we have chosen (or, are given) a chaining input h. We then choose arbitrary values of the first k bytes of the last column of C (excluding the input row containing $h \oplus m$), i.e., the bytes C_i^{15} for $i = 1, \ldots, k$. The value of k will be fixed later (but $1 \leq k \leq 17$).

Having fixed these k bytes of C, we are able to compute the first $k+1$ rows of A, because h is known. We also choose the first k bytes of the last column of B arbitrarily (again, excluding the input row containing m). These are the bytes B_i^{15}, $i = 1, \ldots, k$. Now the values seen in Figure 4.5 are known (for the example value 4 of k)

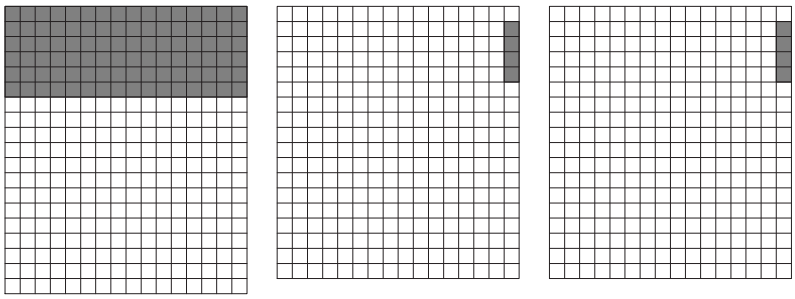
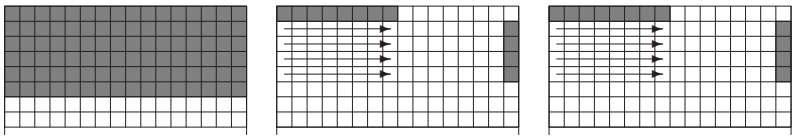


Figure 4.5: Known values in the beginning of the collision attack.

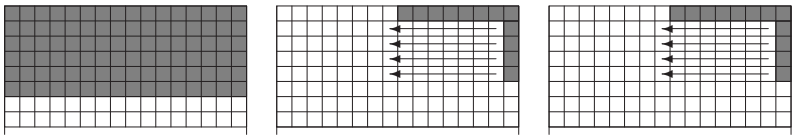
We proceed by performing a meet-in-the-middle attack (see Section 4.1.4) on the first k bytes of column 7 (the middle column) in B and C. Below, ℓ is a number between 0 and 64 which (like k) will be fixed later. T_{L} and T_{R} are hash tables.

1: **for** $i = 0$ to $2^\ell - 1$ **do**
2: Choose an arbitrary 8-byte message M_{L}^i, and let this message be the first half of some message block.
3: Given M_{L}^i, fold right in rectangles B and C as far as possible, i.e., to column 7 of B and C (see Figure 4.6a).
4: Store M_{L}^i in table T_{L}, indexed by the $2k$ bytes of column 7 of B and C, that we just computed. I.e., if these $2k$ bytes are collectively denoted Y^i, do $T_{\mathrm{L}}[Y^i] \leftarrow M_{\mathrm{L}}^i$.
5: Similarly, choose an arbitrary 8-byte message M_{R}^i, and let this message be the last half of some message block.
6: Given M_{R}^i, fold left in rectangles B and C as far as possible, i.e., (again) to column 7 of B and C (see Figure 4.6b).

7: Denote by Z^i collectively the $2k$ bytes of column 7 of B and C, and do $T_{\mathrm{R}}[Z^i] \leftarrow M^i_{\mathrm{R}}$.
8: **end for**
9: Find (in linear time) all indices for which there is an entry in both T_{L} and T_{R}. If the entries are denoted M^i_{L} and M^j_{R}, store the message $M^i_{\mathrm{L}} \| M^j_{\mathrm{R}}$ in table T. The expected number of messages in T is $2^{2\ell-16k}$, since $2^{2\ell}$ pairs $(M^i_{\mathrm{L}}, M^j_{\mathrm{R}})$ can be formed, and each pair constitutes a match on the $2k$ bytes with probability about $2^{-8\cdot 2k}$.
10: Find all collisions $f(h, m) = f(h, m^*)$, where $m, m^* \in T$.



(a) Line 3 of the attack.



(b) Line 6 of the attack.

Figure 4.6: The meet-in-the-middle part of the collision attack.

We may choose k and ℓ so as to optimise the efficiency of the attack. The $2^{2\ell-16k}$ messages in T can be paired up in about $2^{4(\ell-8k)-1}$ ways. Each pair collides with probability about $2^{-8\cdot(17-k)}$, since there are $17-k$ remaining bytes (e.g., the last $17-k$ bytes of the first column in A) on which the messages must agree. Hence, if $2^{4(\ell-8k)-1} > 2^{8(17-k)}$, then we would expect a collision. Thus, we should choose k and ℓ such that

$$4(\ell - 8k) - 1 = 8(17 - k) \iff 4\ell - 24k = 137 \tag{4.2}$$

(with equality in order to find just one collision).

The complexity of the attack is dominated by the loop, i.e., the construction of T_{L} and T_{R}, and the subsequent construction of and search through T (lines 9–10). T_{L} and T_{R} are constructed using $2 \cdot 2^{\ell}$ computations of the $2k$ bytes in column 7 of B and C. Each computation (on average) corresponds to about $2k$ 8-bit XORs, since given, e.g., $m^0, \ldots, m^6$, one may pre-compute part of column 6 and loop through all values of m^7. Then, only a single XOR is required to compute each byte in column 7. On a 32-bit machine, 4 XORs can be done in one operation. Hence, each computation may be estimated to be equivalent to about $2/832 < 2^{-8}$ compression function evaluations. The

exact complexity depends on k, but for values up to $k = 6$, this estimate seems reasonable. Hence, the construction of T_{L} and T_{R} takes time equivalent to about $2^{\ell-7}$ compression function evaluations. The subsequent search through the two lists (line 9) takes a similar amount of time, so the total complexity of lines 1–9 is about $2^{\ell-6}$.

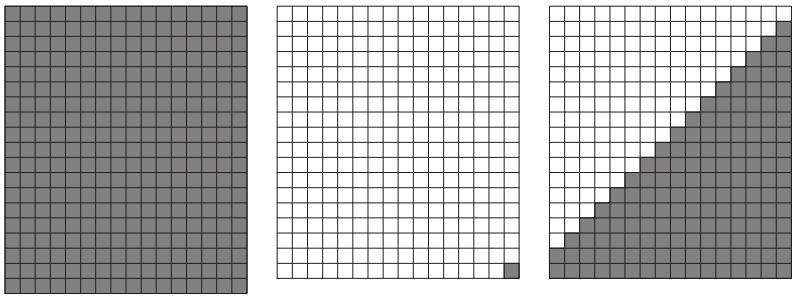


Figure 4.7: Colliding messages will agree on the shaded values (given that they agree on C_1^{15}, and that the chaining input is fixed).

To find collisions in f using messages from T (line 10) one might do as follows. First, notice that a colliding pair will agree on the shaded bytes in Figure 4.7. Using this fact, we may compute for each message in T the byte $C_{k+1}^{16-(k+1)}$; this requires the computation of one row of B, and part of a row of C. We may then place messages in 256 different bins depending on the value of the byte $C_{k+1}^{16-(k+1)}$. In each bin, all messages agree on row $k+2$ of A, so we only have to compute this row once for each bin. We continue, treating each bin separately, and find matches on the byte $C_{k+2}^{16-(k+2)}$. We split each bin into 256 smaller bins based on the value of this byte. Continuing like this, there will be more and more bins, but eventually many of them will contain at most one message; these bins can be discarded. On average, we expect that at most half the state must be computed per message, before the message can be discarded. Hence, the complexity of finding collisions in T may be estimated to about $|T|/2 = 2^{2\ell-16k-1}$ compression function evaluations. Optimally, this should be equal to $2^{\ell-6}$, the approximate complexity of constructing T_{L} and T_{R}. Hence, we look for k and ℓ such that

$$\ell - 6 = 2\ell - 16k - 1 \iff \ell - 16k = -5.$$

Combining this with (4.2) we get

$$40k = 157 \iff k = 3.925.$$

Since k must be an integer, we may choose $k = 4$ and get $\ell \geq 58.25$ from (4.2). With $\ell = 58.25$, the complexity of the attack is about 2^{53}, dominated by the

search through T and with probability of a collision about $1 - 1/e = 0.63$. This complexity is well below the complexity of a standard birthday attack. Increasing ℓ slightly improves the probability of success, but also adds to the complexity (for instance, with $\ell = 59$, the complexity is about 2^{54}, and the probability of success very close to 1). With $k = 4$, the memory requirements are about 2^ℓ message blocks.

Extending to the full hash function

The collision attack just described can be extended into an attack on the full MD2 hash function. There are, however, two complications: (1) padding for the message must be correct, and (2) the checksum block must be correct. Condition (1) is easily fulfilled; simply append to every message an additional message block of 16 bytes each having the value '16'. Condition (2) on the other hand is not as easily dealt with.

By Joux's method (see Section 2.2.3), using 65 collisions with chosen chaining input we can construct a 2^{65}-collision of messages of 65 blocks each. With probability about $p_1 = 1 - 1/e^2 \approx 0.86$, one of the pairs also collides on the checksum block. To find this pair, we need to compute the 2^{65} checksums. Since updating the checksum with one message block takes about $16/832 \approx 2^{-5.7}$ times the time of one compression function evaluation, computing the 2^{65} checksums takes time about $(2^{65} + 2^{64} + \ldots + 2) \cdot 2^{-5.7} \approx 2^{60.3}$ in terms of compression function evaluations. The memory requirements are 2^{65} checksums (along with some bits identifying the combination of message blocks). Finding the $65 \approx 2^6$ collisions takes time about $2^{6+54} = 2^{60}$ using the collision attack just described. Hence, the total complexity of the attack is about $p_1^{-1}(2^{60.3} + 2^{60}) \approx 2^{61.4}$.

To reduce memory requirements, a collision in the checksum can be found by using, e.g., Floyd's cycle-finding algorithm [67]. (Note, however, that the memory requirements for finding the collisions in the compression function are about 2^{59} hash values.) Pre-compute all checksums of the, e.g., 2^{33} combinations of the first 33 message blocks. Use the cycle-finding algorithm to iterate through the full checksums, requiring the computation of $32 = 2^5$ "atomic" checksums for each, hence requiring the time of about $2^{-0.7}$ compression function evaluations. Assuming 2^{65} checksums are needed before the algorithm starts repeating, this method takes time about $2^{64.4}$, which is slightly faster than the generic birthday attack. Other time/memory tradeoffs provide faster methods that require more memory, e.g., about 2^{61} memory and $2^{61.5}$ time. These estimates are to be compared to the standard birthday attack requiring about 2^{64} memory and 2^{65} time (because a message is at least two message blocks including checksum), or the cycle-finding method

applied on the compression function requiring almost negligible memory and time somewhat more than 2^{65} (in the attack above we have assumed that the cycle-finding algorithm requires a factor $\sqrt{2}$ more time than the simple method of storing everything, and with this assumption the cycle-finding algorithm on the full MD2 hash function would take time about $2^{65.5}$).

4.2.3 The preimage attack

We observed that one of the preimage attacks on the MD2 compression function described by Muller [137] in fact allows the attacker more freedom than expected, and therefore it can be used in a preimage attack on the full MD2 hash function.

First, we describe the preimage attack on the compression function.

A preimage attack on the compression function

The preimage attack on the compression function allows the attacker to choose both h and m. We assume the target image, i.e., the required output of the compression function, is h_{T}.

Given h_{T} we can compute the lower right triangle of A by folding up as indicated in Figure 4.8. By arbitrarily fixing A_1^{15} and A_2^{15}, we may complete

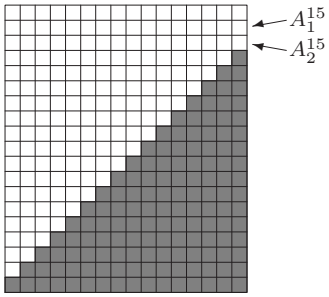


Figure 4.8: The part of A that can be computed given the target image h_{T}.

a further two "diagonals" in A by folding left, see Figure 4.9. We note that fixing A_1^{15} and A_2^{15} introduces a condition on h: one byte-degree of freedom is lost because A_1^{15} depends only on h. A_2^{15}, on the other hand, depends also on the message block, so fixing this byte introduces no condition on h. When h_{T}, A_1^{15} and A_2^{15} are fixed, the rectangle B does not depend on h, but only depends on the message block m.

On the other hand, using just the chaining input h, we can compute all of A by folding left. By fixing the last k bytes of every message block, we can furthermore, independently of the remaining bytes in the message blocks,

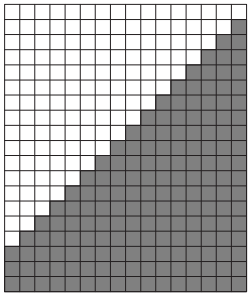


Figure 4.9: The part of A that may be computed given h_{T}, after choosing A_1^{15} and A_2^{15}.

compute a large part of C and part of the last column of B by folding left. See Figure 4.10, which shows how much of B and C can be computed when $k = 6$ is assumed. The blackened bytes play a certain role in the attack. With any given k, we can compute $k + 1$ bytes of the last column of B.

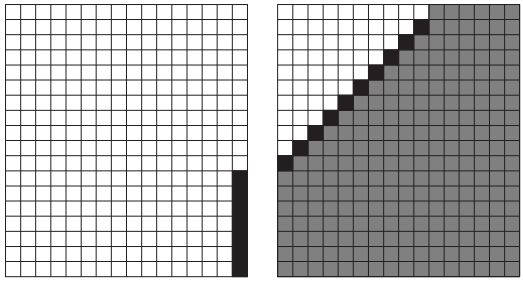


Figure 4.10: The shaded and blackened values of B and C can be computed once the chaining input h and the last 6 bytes of the message block m are fixed.

The idea of the attack is now to compute B by folding right for many different values of the message block m, but where the last k bytes of m are fixed to some constants. Similarly, we compute A and the part of B and C seen in Figure 4.10 for many different chaining inputs h, using the fact that we know the last k bytes of the message block. We then look for a collision on the blackened bytes of B in Figure 4.10. For each collision, we check if the message block and the chaining input also match on the remaining blackened bytes (those in C) in Figure 4.10. If they match, then we have found a chaining input h and a message block m such that $f(h, m) = h_{\mathrm{T}}$.

An algorithmic version of the attack, with any value of k from 0 to 16, follows. Assume that we are given the target chaining output h_{T}.

1: Choose A_1^{15} and A_2^{15} arbitrarily, and compute the part of A that is shaded in Figure 4.9.

2: **for** $i = 1$ to $2^{8(16-k)}$ **do**
3: Choose an arbitrary message block m^i such that the last k bytes are fixed to, say, zeroes.
4: Given m^i, compute B by folding right. Let Y^i denote the last column of B, and do $T_1[Y^i] \leftarrow m^i$.
5: **end for**{See Figure 4.11}
6: **for** $i = 1$ to $2^{8(k+1)}$ **do**
7: Choose an arbitrary chaining input h^i such that A_1^{15} as chosen in line 1 is correct (note that this byte depends only on the chaining input).
8: Given h^i, compute all of A, and the parts of B and C that are shaded in Figure 4.10 (all by folding left). Denote by Z^i collectively the bytes that are blackened in Figure 4.10, and do $T_2[Z^i] \leftarrow h^i$.
9: **end for**
10: Find collisions (h, m) in indices of T_1 and T_2, restricted to the last $k+1$ bytes of the last column of B, i.e., the blackened bytes of B in Figure 4.10. Since T_1 contains $2^{8(16-k)}$ values, and T_2 contains $2^{8(k+1)}$ values, and the collision must occur in $k+1$ bytes, we expect to find about $2^{8(16-k)+8(k+1)-8(k+1)} = 2^{8(16-k)}$ collisions.
11: **for** each collision (h, m) **do**
12: Check if the pair (h, m) also agrees in the remaining $16-k$ blackened bytes in Figure 4.10. This happens with probability about $2^{-8(16-k)}$.
13: **end for**

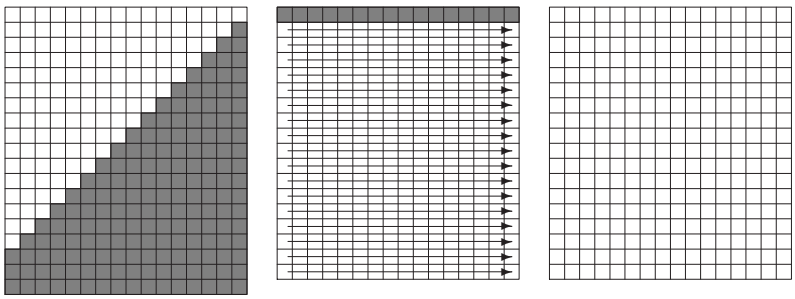


Figure 4.11: Lines 2–5 of the attack.

Since we find an expected $2^{8(16-k)}$ collisions in line 10, we expect to find one preimage with this method.

Let us find the optimal value of k. In the first loop (lines 2–5), we compute B $2^{8(16-k)}$ times. Since computing B corresponds to evaluating about one third of the compression function, the first loops takes time below $2^{8(16-k)-1}$. The second loop (lines 6–9) requires that half of A, most of C and $k+1$ bytes of B be computed, $2^{8(k+1)}$ times. The complexity is equivalent to about

$2^{8(k+1)-1}$ evaluations of the compression function. In line 10 we find collisions between T_1 and T_2. Since the tables are sorted, the expected complexity of finding the collisions is about $\max(2^{8(16-k)}, 2^{8(k+1)})/832$, assuming that a comparison on the $k+1$ bytes on average requires an amount of work similar to computing one byte in the compression function (recall that 832 bytes need to be computed in the compression function). In the last loop (lines 11–13) we need to check each collision to see if the collision extends to the remaining blackened bytes. There are about $2^{8(16-k)}$ collisions, and we need to compute on average about $16-k$ bytes to check if there is a match (in most cases, there will be no match already on the byte C_1^{15-k}, i.e., the top blackened byte in Figure 4.10). Hence, the expected complexity of the last loop is about $2^{8(16-k)} \cdot (16-k)/832$.
To sum up, the three loops are expected to dominate the total complexity, so we try to minimise these complexities. Having $16-k = k+1$ would give the first two loops the same complexity. This would result in k being a non-integer, which complicates the attack. With $k = 7$ we get a total complexity of about $2^{71} + 2^{63} + 2^{72} \cdot 9/832 \approx 2^{71}$. With $k = 8$ the complexity is about $2^{63} + 2^{71} + 2^{64} \cdot 8/832 \approx 2^{71}$.

We would like to point out that there are two important differences between the description of the attack given here, and that of [137]:

- In [137], k is chosen to be 6, but the work done in the first two loops is scaled differently. One does not really need to choose $2^{8(16-k)}$ message blocks and $2^{8(k+1)}$ chaining inputs, as described above, but one cannot choose more message blocks than this number, and the product of the two numbers must be at least 2^{136} to get a preimage with good probability. The resulting complexity in [137] is dominated by the last loop, and is therefore about $2^{80} \cdot 10/832 \approx 2^{73.6}$.

- In [137], the last $k = 6$ bytes of h are also fixed. This is not necessary, since C can be (and needs to be) computed from scratch for every value of h. The reason is that the first column of A depends on all bytes of h, and therefore C does as well because we compute C by folding left. The fact that only a single byte of h is fixed (by A_1^{15}) leads to an improvement of the preimage attack on the full MD2 hash function, as we shall see in the following.

Extending to the full hash function

In the previous papers [101, 137] containing preimage attacks on MD2, a different type of preimage attack on the compression function than the one described above was used. This was because the attack described above

placed strong restrictions on the chaining input h. However, given the additional freedom in the choice of h provided by the attack described above, it can be used much more efficiently to construct preimages for the full hash function.

The idea is to compute the $2^{8(k+1)}$ chaining inputs from a given initial value, instead of choosing them directly. Say we are given the initial value h. We deliberately do not denote the initial value by h_0, for reasons which will become clear later. Then we may choose $2^{8(k+2)}$ message blocks m_1 and compute $h_1 = f(h, m_1)$ for each value of m_1. Since only about $1/256$ of these will produce the right value of A_1^{15} (which we choose in the beginning, according to the attack on the compression function), there will be about $2^{8(k+1)}$ valid chaining inputs h_1, which can be used in the attack described above (note that the number of chaining inputs matches the number required in the attack).

We still have not taken the checksum into account. We shall postpone this a little longer, and determine the complexity of the attack when the chaining inputs are computed rather than chosen directly. The only difference is in the second loop (lines 6–9), where the complexity now is about $2^{8(k+2)}$ instead of $2^{8(k+1)-1}$. With $k = 7$ the total complexity (still ignoring the checksum) is about $2^{71} + 2^{72} + 2^{72} \cdot 9/832 \approx 2^{72.6}$.

The attack described provides two messages blocks, say m_1 and m_2, such that $h_1 = f(h, m_1)$ and $h_{\mathrm{T}} = f(h_1, m_2)$ for some given target image h_{T}, and a given initial chaining value h. The probability that m_2 is the checksum of m_1, as required, is only about 2^{-128}. However, we do not have to carry out the attack 2^{128} times. Instead, we do the following (assume we are given target hash value h_{T}).

1: Starting from the initial value of MD2, produce a 2^{128}-collision by the method of Joux (see Section 2.2.3). Let the common chaining output of all these 2^{128} messages be h_{128}.
2: With $h = h_{128}$, carry out the preimage attack described above. This preimage attack produces two message blocks m_1 and m_2 such that $f(h, m_1) = h_{129}$ and $f(h_{129}, m_2) = h_{\mathrm{T}}$.
3: Compute the checksum state v^* such that $c(v^*, m_1) = m_2$, where c is the checksum function of MD2.
4: Perform a meet-in-the-middle search through the 2^{128}-collision to find a message M in this multi-collision which has checksum v^* (recall that every message in the multi-collision produces the same intermediate hash value h_{128}).
5: **return** $m^* = M \| m_1$ $\{m_2$ is the checksum of $m^*\}$

We note that the idea of using a 2^{128}-collision as a prefix to the mes-

sage block m_1 came from a series of generic attacks on checksum-based hash functions (see [69] and Section 4.4).

In line 1 we need to find 128 collisions in the MD2 compression function. By the birthday method, this has complexity about $2^{64+7} = 2^{71}$. By using the collision attack described previously, the complexity is only about 2^{61}. Line 2 has complexity about $2^{72.6}$, as found above. Line 3 has negligible complexity, and line 4 requires about 2^{66} evaluations of the checksum function, when the tree structure of the multi-collisions is taken into account. Hence, the complexity is about $2^{60.3}$ in terms of compression function evaluations. We note that the message block m_1 must contain proper padding, but this can easily be ensured in line 2. Altogether, line 2 dominates the attack with respect to complexity, and hence the total complexity is about $2^{72.6}$. If collisions are found by a birthday attack in line 1, then the total complexity is about 2^{73}.

The preimage is of length 129 message blocks. If we can find a shorter 2^{128}-collision, then we can find a shorter preimage. We now describe how the preimage attack on the compression function can be used to produce short multi-collisions faster than by brute force. Then we explain how to use these multi-collisions to produce shorter preimages for the full hash function.

Multi-collisions

By using more chaining inputs in the preimage attack on the compression function, we can find several preimages. For instance, we may choose 2^{80} message blocks m_1, used to produce chaining inputs h_1 for the attack. This would result in about 2^{72} valid chaining inputs. With $k = 7$ we get an expected 2^8 preimages. The complexity is about 2^{80} in terms of compression function evaluations. Notice that this attack can be carried out using any initial chaining value h, and any target chaining value h_{T}. The result is a set of 2^8 two-block messages all mapping h to h_{T}, and hence we have a 2^8-collision. This method can be generalised; a 2^t-collision can be constructed in time about 2^{72+t} for t from 0 to 56.

Shorter preimages

Let $h_{32} = h_{\mathrm{T}}$ be the target hash, and choose h_{30} arbitrarily. Then find a 2^8-collision mapping h_{30} to h_{32}. Repeat this procedure, finding 2^8-collisions mapping h_{2i} to $h_{2(i+1)}$ for decreasing i from 14 down to 0. h_{2i} may be chosen arbitrarily in each step, except when $i = 0$: h_0 must be chosen as the initial value of MD2.

By this procedure we obtain $(2^8)^{16} = 2^{128}$ preimages of h_{T}, and we expect

one of them to have the right checksum. The complexity is about $16 \times 2^{80} = 2^{84}$, and the preimages are of length $2 \cdot 16 - 1 = 31$ blocks (the last block being the checksum). Other combinations of preimage lengths and complexities are possible; see Table 4.1 for examples.

Table 4.1: Complexity of the preimage attack with different preimage lengths.

Preimage length (blocks)	Complexity
129	$2^{72.6}$
63	2^{81}
31	2^{84}
15	2^{91}
7	2^{106}

4.2.4 Second preimages

The preimage attack described above is also a second preimage attack, since it is trivial to ensure that the preimage obtained is different from some given first preimage. Hence, second preimages of length 129 blocks can be found in time about $2^{72.6}$ (the length of the first preimage is arbitrary).

4.2.5 Summary

In this section, we described a collision attack on MD2 of complexity about $2^{61.4}$, which is to be compared to the complexity of a birthday attack of about 2^{65} (since the compression function is evaluated at least twice for every message, due to the checksum block). We also described a preimage attack of complexity about $2^{72.6}$, which is far below the complexity of the brute force preimage attack (2^{129}), and also well below the complexity of the previous best known preimage attack [101] of about 2^{97}.

4.3 Cryptanalysis of MDC-2

As described in Section 3.1, MDC-2 is a method of constructing hash functions from block ciphers, where the output size of the hash function is twice the size of the block cipher. MDC-2 was developed at IBM in the late 80s. A conference paper by IBM researchers Meyer and Schilling from 1988 describes the construction [135]. A patent was filed in August 1987, and the patent was issued in March 1990 [26]. The construction was standardised in

ISO/IEC 10118-2 in 1994 [86]. It is mentioned in great detail in both the Handbook of Applied Cryptography [131, Alg. 9.46] and in the Encyclopedia of Cryptography and Security [210, pp. 379–380]. Furthermore, it is in practical use (see e.g., [88, 112, 200]).

Since publication, there seems to have been a wide belief in the cryptographic community that given an ideal block cipher, MDC-2 provides a collision resistant hash function. By this we mean that given an n-bit block cipher (thus yielding a $2n$-bit hash function), the required effort to find a collision in the hash function is expected to be 2^n. However, there is no proof of this property. The only proof that collision resistance is better than $2^{n/2}$, as offered by many simpler (single-length) constructions, is due to Steinberger [197], who showed that for MDC-2 based on an ideal cipher, an adversary asking less than $2^{3n/5}$ queries has only a negligible chance of finding a collision.

This section describes unpublished cryptanalytic results on MDC-2 [103]. We provide the first collision attack on MDC-2 which breaks the birthday bound. The attack makes no non-standard assumptions on the underlying block cipher. When applied to an instantiation of MDC-2 with e.g., a 128-bit block cipher (see e.g., [212]), the attack has complexity about $2^{124.5}$, which is better than the expected 2^{128} collision resistance for an ideal 256-bit hash function.

We also present improved preimage attacks on MDC-2. The previous best known preimage attack, first described by Lai and Massey [115], has time complexity about $2^{3n/2}$ and requires around $2^{n/2}$ memory. In this section, we provide a range of time/memory trade-offs, the fastest of which is significantly faster than the Lai/Massey attack. We describe attacks of any time complexity from 2^n to 2^{2n}. The memory requirements are such that the product of the time and space complexities is always around 2^{2n}. Hence, our most efficient preimage attack has time and space complexity about 2^n.

4.3.1 Preliminaries

Recall Definition 2.3 of multi-collisions, and the expected complexity of finding such multi-collisions (see Section 2.2.3). Multi-collisions are used in the collision attack on MDC-2.

MDC-2 was originally defined using DES [140] as the underlying block cipher. Here, we think of MDC-2 as a general double-length construction method for hash functions based on block ciphers. For ease of presentation, we shall assume that keys and message blocks are of the same size, even if this is in fact not the case for DES.

For a description of MDC-2, we refer to Section 3.1 (Construction 3.5).

We should add that in the original description of MDC-2 [135], two bits of each of the two keys h_{i-1} and $\tilde{h}_{i-1}$ were fixed. This had two implications. First of all, all known weak and semi-weak keys of DES were ruled out, and secondly, this measure ensured that the two keys were always different. There seems to be no strong consensus that fixing key bits is a necessary security measure when MDC-2 is based on some other block cipher for which weak keys are not believed to exist. However, one might argue that ensuring that the two keys are different increases security – although this practice also has a cost in terms of security: the amount of state passed on from one iteration to the next is less than $2n$ bits. The attacks presented here can be applied regardless of whether or not some key bits are fixed.

A generalisation.

We may generalise the MDC-2 construction. Let $f : \{0,1\}^n \times \{0,1\}^n \rightarrow \{0,1\}^n$ be an arbitrary function, and let g be any (efficiently invertible) bijection from $2n$ bits to $2n$ bits. Then, a generalised construction is the following.

$$\begin{aligned} W &= f(h_{i-1}, m_i) \| f(\tilde{h}_{i-1}, m_i) \\ h_i \| \tilde{h}_i &= g(W). \end{aligned} \tag{4.3}$$

See Figure 4.12. The attacks presented here apply to any instance of this

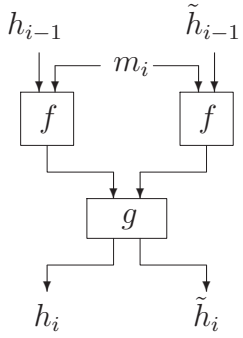


Figure 4.12: The generalised MDC-2 construction.

construction. Notice that MDC-2 has $f(x, y) = E_x(y) \oplus y$ and $g(a\|b\|c\|d) = a\|d\|c\|b$. In the following, we shall use the notation of the generalised construction. We assume that evaluating g (both forwards and backwards) costs much less than evaluating f. Our complexity estimates will be in terms of compression function evaluations. For example, if an attack requires T calls of f, we shall count this as having time complexity $T/2$.

4.3.2 The collision attack

The collision attack applies to any construction of the type (4.3), where the outputs of f are (roughly) Poisson distributed. This is indeed expected to be the case when f is instantiated by $E_K(m) \oplus m$, where E is the encryption function of an ideal block cipher.

In the attack, one first finds by the brute force method, starting from the initial value of the hash function, an r-collision in the left chain of Figure 4.12, after the application of the function g (i.e., an r-collision in h_1). Then, one searches for a message block such that two different inputs out of the r possibilities in the right chain of the next iteration collide. Since the left chain already contains a collision, the collision now spans both chains. An algorithmic version of the attack follows.

1: **repeat**
2: Choose a random message block m_1
3: $h_1 \| \tilde{h}_1 \leftarrow g(f(h_0, m_1) \| f(\tilde{h}_0, m_1))$
4: **until** an r-collision $(m_1^1, \ldots, m_1^r)$ in h_1 has been found
5: Denote the corresponding r values of $\tilde{h}_1$ by $\tilde{h}_1^1, \ldots, \tilde{h}_1^r$.
6: **loop**
7: Choose a random message block m_2
8: **for** $j = 1$ to r **do**
9: $U_j \leftarrow f(\tilde{h}_1^j, m_2)$
10: **end for**
11: **if** $U_i = U_j$ for some (i, j), $i \neq j$ **then**
12: **return** $(m_1^i \| m_2, m_1^j \| m_2)$
13: **end if**
14: **end loop**

See Figure 4.13. The first loop (lines 1–4) is expected to take time $q_1 = (r! 2^{n(r-1)})^{1/r}$ (cf. Section 2.2.3). The probability that the condition in line 11 is fulfilled is about $\binom{r}{2} 2^{-n}$, since there are $\binom{r}{2}$ pairs of n-bit values, which must be compared. Hence, we expect to go $2^n / \binom{r}{2}$ times around the final loop (lines 6–14). In each iteration of the loop, f is evaluated r times. In the construction (4.3), f is evaluated twice per message block, and hence the r evaluations of f are equivalent to $r/2$ compression function evaluations. The total work required in the final loop is therefore expected to be

$$q_2 = (r/2) \cdot 2^n / \binom{r}{2} = 2^n/(r-1).$$

The total work required by the attack is $q_1 + q_2 = (r! 2^{n(r-1)})^{1/r} + 2^n/(r-1)$. Hence, we may choose r as the integer ≥ 2 that minimises this expression. Notice that q_1 is an increasing function of r, and q_2 is decreasing. By setting

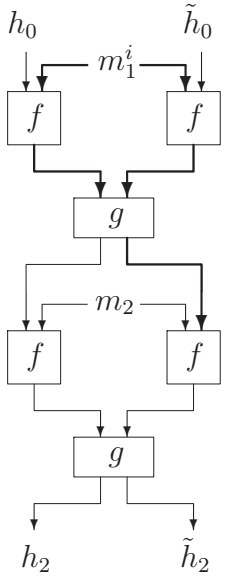


Figure 4.13: The collision attack on MDC-2. Thick lines mean that there are r different values of this variable. Thin lines mean that there is only one.

$q_1 = q_2$ one gets, very roughly, a time complexity around $(\log_2(n)/n)2^n$. However, it turns out that the best choice of r is not exactly the one where $q_1 = q_2$, as one might expect. The easiest way to find the optimal value of r for a given n seems to be to simply try increasing values from $r = 2$ until the complexity starts to increase. The results of this method can be found in Table 4.2, that shows the best choices of r and the corresponding complexities for different sizes n of the block cipher.

Table 4.2: Time complexity of the collision attack on MDC-2 with an n-bit block cipher, compared to birthday complexity.

n	r	Collision attack complexity	
		Section 4.3.2	Birthday
64	9	$2^{61.3}$	2^{64}
128	14	$2^{124.5}$	2^{128}
256	24	$2^{251.7}$	2^{256}

The probability of success of our attack with these complexities is about $1-1/e$ for Step 1, and the same probability for Step 3 when repeated $2^n/\binom{r}{2}$ times, in total $(1-1/e)^2 \approx 0.40$. The probability of success for the birthday attack with 2^n queries is about $1 - e^{-1/2} \approx 0.39$. Hence, we consider the

comparisons fair.

4.3.3 Preimage attacks

A brute force preimage attack on MDC-2 (or on (4.3) in general) has time complexity $O(2^{2n})$ and space complexity $O(1)$. The previous best known preimage attack is due to Lai and Massey [115], and has time complexity $O(2^{3n/2})$ and space complexity $O(2^{n/2})$. Hence, for both attacks the product of the time complexity and the space complexity is $O(2^{2n})$. In the following subsection, we describe a generalised attack for which the product of the time and the space complexities is at most $n2^{2n}$, but where time complexity can be anything between $O(n2^n)$ and $O(2^{2n})$. We then describe how to reach a time and space complexity of $O(2^n)$.

An attack allowing for time/memory trade-offs.

The outline of the attack is as follows.

1: Build a binary tree of pseudo-preimages with the target image $h_{\mathrm{T}}\|\tilde{h}_{\mathrm{T}}$ as root: the nodes are labelled with intermediate hash values, and each edge is labelled with a message block value meaning that this message block maps from the intermediate hash value at the child node to the intermediate hash value at the parent. The tree has (on average) two children for each node, and it has depth d, which means that there are 2^d leaves.

2: From the initial value $h_0\|\tilde{h}_0$ of the hash function, find a message block that produces an intermediate hash value equal to one of the leaves in the tree from Step 1.

See Figure 4.14. This technique clearly leads to a preimage consisting of a

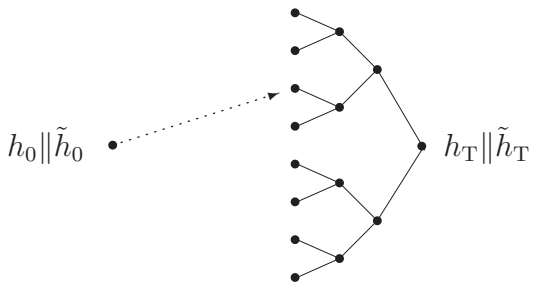


Figure 4.14: A binary tree of pseudo-preimages of depth $d = 3$.

message block that maps to a leaf ℓ in the tree, and a sequence of d message

blocks corresponding to the path in the tree that leads from the leaf ℓ to the root. Hence, the total length of the message is $d+1$ blocks.

The value of d determines the time/memory trade-off. We shall discuss concrete values of d later. The cost of Step 1 will be evaluated in the following. Since the tree has 2^d leaves, Step 2 is expected to take time 2^{2n-d}. In effect, by constructing the tree we produce 2^d new target images, which improves the efficiency of the final brute force search by a factor of 2^d. The memory requirements are $2^d + 2^{d-1} + \ldots + 1 \approx 2^{d+1}$ intermediate hash values.

We note that the last message block, the one that maps to the target image, must contain proper padding for a message of $d+1$ blocks. If there are not enough degrees of freedom in the last block to both ensure proper padding and to find two pseudo-preimages, then a few initial steps are needed to ensure proper padding. It will become clear in the following that this only has a small effect on the total time complexity.

Constructing the tree (Step 1 above) is a time-expensive task for an ideal hash function. However, for the MDC-2 construction, it can be done efficiently.

Theorem 4.1. *Given a target hash value $h_{\mathrm{T}}\|\tilde{h}_{\mathrm{T}}$, a pseudo-preimage can be found in time at most 2^{n-1} (in terms of compression function evaluations) with probability about $(1-1/e)^2$. By a pseudo-preimage we mean a pair $(h_{\mathrm{p}}, \tilde{h}_{\mathrm{p}})$ and a message block m such that $g(f(h_{\mathrm{p}}, m)\|f(\tilde{h}_{\mathrm{p}}, m)) = h_{\mathrm{T}}\|\tilde{h}_{\mathrm{T}}$.*

Proof. The method is the following. Let $U\|\tilde{U} = g^{-1}(h_{\mathrm{T}}\|\tilde{h}_{\mathrm{T}})$. Choose m arbitrarily, define $f_m(x) = f(x, m)$, and find by brute force preimages h_{p} and $\tilde{h}_{\mathrm{p}}$ of U and $\tilde{U}$, respectively, under f_m. This means that $g(f_m(h_{\mathrm{p}})\|f_m(\tilde{h}_{\mathrm{p}})) = h_{\mathrm{T}}\|\tilde{h}_{\mathrm{T}}$.
Assuming that the outputs of f_m are Poisson distributed, the probability that a given image has at least one preimage is $1-1/e$. Hence, the probability of finding a preimage of both U and $\tilde{U}$ is $(1-1/e)^2$. A brute force search for preimages on an n-bit function has complexity at most 2^n, which in this case corresponds to 2^{n-1} compression function evaluations. □

We note that for an ideal $2n$-bit compression function, the above task has complexity about 2^{2n}. The story does not finish with Theorem 4.1, however. Clearly, by evaluating a random n-bit function f 2^n times, one finds on average one preimage for all elements of $\{0,1\}^n$. Thus, we obtain the following corollary.

Corollary 4.1. *Given t target hash values, in time 2^{n-1} one pseudo-preimage (on average) can be found for each target hash value. Here, t can be any number from 1 to 2^n.*

Proof. The technique is the same as above (we note that inverting g, which must be done 2^t times, is assumed to be a much simpler task than evaluating f). Since f_m is evaluated on all 2^n possible inputs, on average one preimage is found for each element of $\{0,1\}^n$. Therefore, we also expect one preimage on average of each of the t target hash values. □

We note that in the case of MDC-2, where g has a special form that allows to compute n bits of the output given only n bits of the input (and vice versa), t above can actually be 2^{2n} without affecting the complexity. The reason is that g (in this case) never has to be inverted more than 2^n times.

Due to Theorem 4.1 and Corollary 4.1, the tree described above can be efficiently constructed as follows:

1: The tree initially contains only the root, labelled $h_{\mathrm{T}} \| \tilde{h}_{\mathrm{T}}$
2: Using the method of Theorem 4.1 twice, find two pseudo-preimages of the root, and add these to the tree as children of the root
3: **for** $i = 2$ to d **do**
4: Using the method of Corollary 4.1 (twice), find two pseudo-preimages of each leaf in the tree, and add these to the tree
5: **end for**{The tree now has depth d and 2^d leaves}

The expected time required in line 2 is 2^n. The expected time required in each iteration of the loop (lines 3–5) is 2^n. Hence, the total time complexity is $d2^n$.

As mentioned, with 2^d leaves, meaning 2^d new target images, finding by brute force a true preimage has complexity 2^{2n-d}. Hence, the total time complexity is about $d2^n + 2^{2n-d}$. As mentioned above, memory requirements are about 2^{d+1}.

Observe that with $d = 0$ one gets time complexity 2^{2n} and space complexity 1, which is not surprising since we do not build a tree at all, so we have a standard brute force preimage attack. With $d = n/2$ one gets time complexity about $2^{3n/2}$ and space complexity about $2^{n/2}$, equivalent to the attack of Lai and Massey. The most efficient attack appears when $d = n$, in which case the time complexity is about $(n+1)2^n$, and the space complexity is 2^{n+1}. We improve the efficiency of this particular time/memory trade-off below.

We note that this attack provides practically any time/memory trade-off for which the product of the time and the space complexities is about 2^{2n}. Figure 4.15 shows some example trade-offs.

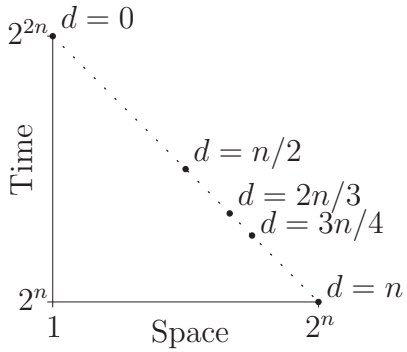


Figure 4.15: A visualisation of the time/memory trade-off. Both axes are logarithmic. The case $d = 0$ corresponds to the brute force attack, and $d = n/2$ corresponds to the previous best known preimage attack of Lai and Massey. Larger values of d constitute improvements with respect to attack efficiency.

Alternative methods.

The tree above does, in fact, not have to be binary. If every node has on average 2^b children, then when the tree has depth d, there are 2^{bd} leaves. The time required to construct the tree is $d2^{b+n-1}$. The time required for Step 2 above is 2^{2n-bd}. The memory requirements are about 2^{bd} for reasonably large b. With $b = n/(d+1)$, which approximately balances the time spent in Steps 1 and 2, the total time complexity is about $(d/2+1)2^{n(d+2)/(d+1)}$ and the memory requirements are $2^{nd/(d+1)}$. With $d \approx n$, the attack is roughly the same as above, but with smaller values of d, the number of children of each node grows.

An alternative way of constructing the tree is the following. First, find a pseudo-preimage of the root. Then, find a pseudo-preimage of the root and its child. Continue applying Corollary 4.1 this way, finding in each step a pseudo-preimage for each node in the tree, thus doubling the tree size in every step. After d steps, the tree contains 2^d nodes. The time complexity is $d2^{n-1}$. See Figure 4.16.

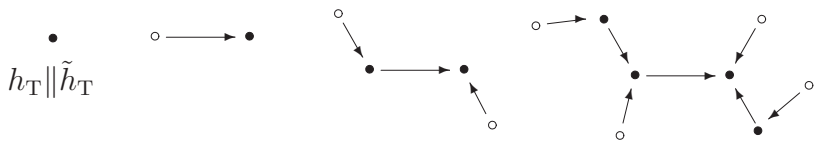


Figure 4.16: Constructing a tree of pseudo-preimages by finding one child of every node in each step.

Now, if there is no length padding, then we may perform a brute force search that links the initial value to any of the 2^d nodes in the tree. This brute force search has complexity 2^{2n-d}. Compared to the variant of the previous section, both time and space requirements are roughly halved. We note that this attack resembles a method described by Leurent [119] of finding preimages of MD4.

Length padding can be circumvented in the same way as it is circumvented in Kelsey and Schneier's second preimage attack on the Merkle-Damgård construction (Section 2.2.4), but the resulting attack is slightly slower than the variant above, since there is (apparently) no efficient method of finding fixed points of the compression function.

Pushing the time complexity down to 2^n.

The attack above can be modified such that it has time complexity very close to 2^n. This attack is similar to a preimage attack by Mendel and Rijmen on the HAS-V hash function [130], and also bears resemblance with the P^{3}graph method introduced by De Cannière and Rechberger [30].

In the attack, two message blocks m_0 and m_1 are first chosen, such that they have correct padding for a message of length $n+1$ blocks. Here, we assume that padding does not fill an entire message block. Before the attack starts, two (initially empty) hash tables U_0 and U_1 with a capacity of 2^n each must be prepared (recall that collisions in the hash tables can be handled with a linked list, but here we ignore this possibility). The attack proceeds by computing $y_i = f(i, m_0)$ for all i from 0 to $2^n - 1$, and placing the outputs in U_0, indexed by y_i. Similarly with m_1, where the outputs are placed in U_1. Then, one constructs a binary tree with the target image at the root, and such that every node has (on average) two children, that can be looked up in U_0 and U_1. When the tree has 2^n leaves, one searches for a message block mapping the initial value to a leaf in the tree. The total expected time is about 2^{n+1}, and the memory requirements are also 2^{n+1}. In algorithmic form, the attack can be described as follows.

1: **for** $i = 1$ to $2^n - 1$ **do**
2: Compute $f(i, m_0)$, and do $U_0[f(i, m_0)] \leftarrow i$
3: Compute $f(i, m_1)$, and do $U_1[f(i, m_1)] \leftarrow i$
4: **end for**
5: Place the target image $h_{\mathrm{T}} \| \tilde{h}_{\mathrm{T}}$ at the root of a binary tree.
6: **for** $i = n + 1$ down to 2 **do**
7: **for** each leaf $h_i \| \tilde{h}_i$ in the tree **do**
8: $G_i \| \tilde{G}_i \leftarrow g^{-1}(h_i \| \tilde{h}_i)$
9: Add $U_0[G_i] \| U_0[\tilde{G}_i]$ to the tree as a child of $h_i \| \tilde{h}_i$

10: Add $U_1[G_i]\|U_1[\tilde{G}_i]$ to the tree as a child of $h_i\|\tilde{h}_i$
11: **end for**
12: **end for**
13: Sort the 2^n leaves $h_1\|\tilde{h}_1$ of the tree by placing them in the table T
14: **loop**
15: Choose a random message block d
16: $h_1\|\tilde{h}_1 \leftarrow g(f(h_0, d)\|f(\tilde{h}_0, d))$
17: **if** $h_1\|\tilde{h}_1 \in T$ **then**
18: **return** d and the path in the tree from $h_1\|\tilde{h}_1$ to the root
19: **end if**
20: **end loop**

We note that if padding spans several message blocks, a few initial steps are required to invert through the padding blocks. This may add a small factor of 2^n to the complexity.

Table 4.3 shows some example complexities of this attack for different sizes of n, compared to the previous best known preimage attack and the brute force attack.

Table 4.3: Time complexities of the preimage attack of Section 4.3.3 compared to the previous best known preimage attack of Lai and Massey, and to a brute force attack.

n	Preimage attack complexity		
	Section 4.3.3	Lai-Massey	Brute force
54	2^{55}	2^{81}	2^{108}
64	2^{65}	2^{96}	2^{128}
128	2^{129}	2^{192}	2^{256}
256	2^{257}	2^{384}	2^{512}

4.3.4 Other non-random properties

Let M be a message of t blocks, and let $H(M) = h_t\|\tilde{h}_t$ be the MDC-2 hash of M. The probability that $h_t \neq \tilde{h}_t$ is $(1-2^{-n})^t$, because the two halves must be different after the processing of every block out of the t blocks, in order for them to be different at the end. For an ideal $2n$-bit hash function, this probability is $1 - 2^{-n}$, irrespective of the value of t. Hence, when $t \gg 1$, the probability of the two output halves being equal is much higher in MDC-2 than in an ideal hash function. In fact, if $t = 2^n$, then the probability is around $1 - 1/e \approx 0.63$, since $(1-2^{-n})^{2^n} \approx 1/e$ for plausible values of n. The

property does not hold for the construction (4.3) in general (nor does it hold if some key bits are fixed to ensure that the two keys in each iteration are different). What is required is that some n-bit value b exists for every n-bit value a such that $g(a\|a) = b\|b$.

If, during the processing of a message, one has obtained two equal halves, then a standard birthday collision attack can be applied in time $2^{n/2}$. Hence, a new type of birthday attack on MDC-2 is as follows. Search for a message block m_0 such that $f(h_0, m_0) = f(\tilde{h}_0, m_0) = h_1$. Then, find a pair (m_1, m_1^*) of message blocks such that $f(h_1, m_1) = f(h_1, m_1^*)$. This attack takes the same amount of time as a standard birthday attack (it is in fact faster by a factor of two, since f only has to be called about 2^n times), but a naive implementation uses only $2^{n/2}$ memory compared to 2^n for a (naive) standard birthday attack. By using cycle-finding methods, memory requirements can be made negligible in both cases.

4.3.5 Application to other constructions

The construction (4.3) can be generalised even further. For example, we may define the following general construction, where f and $\tilde{f}$ are two distinct functions both mapping as $\{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$, and $g : \{0,1\}^{2n} \to \{0,1\}^{2n}$ is (again) an invertible mapping:

$$\begin{aligned} W &= f(h_{i-1}, m_i)\|\tilde{f}(\tilde{h}_{i-1}, m_i) \\ h_i\|\tilde{h}_i &= g(W). \end{aligned} \tag{4.4}$$

Our attacks also apply to this construction, except that in some cases the complexity is up to twice as high. For instance, the attack of Theorem 4.1 now requires 2^n evaluations of both f and $\tilde{f}$, and hence the total time complexity is comparable to 2^n compression function evaluations, and not 2^{n-1} as is the case when $f = \tilde{f}$.

Vortex [79] is a candidate for the SHA-3 hash function competition, which uses the generalised MDC-2 construction (4.3). The function g is based on carry-less multiplications, additions, and XORs. It is not a permutation, and is not efficiently invertible; inversion seems to take time about 2^n, assuming that the size of the image is not much below 2^{2n}, which is a fact that is hard to prove, but looking at small variants of the function, it seems reasonable. This means that the collision attack described in Section 4.3.2 applies to Vortex, and may have lower complexity since collisions are generated not only in f, but also in g.

The preimage attacks described in Section 4.3.3 cannot all be applied directly to Vortex due to g not being efficiently invertible, and due to the

fact that Vortex employs the EMD construction (see Section 3.3.4); however, the preimage attack of Lai and Massey [115] applies to Vortex, and several time/memory trade-offs are possible: preimages can be found in time about 2^{2n-k} using about 2^k memory, where k ranges from 0 to $n/2$.

4.4 Generic attacks on checksum-based hash functions

Checksum-based hash functions in general, described in Section 3.3.3, were first analysed in [69], where linear checksums were shown not to offer much protection against multi-collision attacks, second preimage attacks and the Nostradamus attack (see Section 2.2). Here, we investigate the effect of non-linear checksums. The work presented is published as a technical report [70].

A generic method of circumventing the checksum function is to first construct a multi-collision (see Definition 2.3) of size 2^μ, where the checksum chain is ignored. Then, starting from the output of the multi-collision, one carries out the attack in question – this might be the multi-collision attack, the second preimage attack, the Nostradamus attack (Section 2.2), or any other attack on the standard Merkle-Damgård hash function. At the end, the checksum block will be fixed by the attack – e.g., in the second preimage attack, the checksum block is fixed by the first preimage. From the multi-collision generated in the beginning, a message having the correct checksum can be found; since there are 2^μ messages in the multi-collision, one of them is expected to have the right checksum. The time required to find this message depends on the details of the checksum function.

4.4.1 Invertible checksum function

Assume first that the checksum function c can be inverted in time about the same as the time required to evaluate c. By inverting we mean given y, z to find x such that $c(x, y) = z$. We also introduce the term *bridging*, which means to find y given x and z. For now, we assume that both operations can be done as efficiently as computing c in the forward direction.

To carry out the attack, one first constructs a multi-collision S of size 2^μ, meaning a sequence of length μ blocks of one-block collisions m_i^b, $b \in \{0, 1\}$ and $1 \leq i \leq \mu$, such that $h_i = f(h_{i-1}, m_i^0) = f(h_{i-1}, m_i^1)$, where h_0 is the initial value of the hash function. This multi-collision structure is called a *checksum control sequence* in [69]. It takes time about $\mu 2^{n/2}$ to construct it using the technique described in Section 2.2.3. Recall that we ignore its effect

on the checksum chain for now. The multi-collision produces the (common) intermediate hash value $h_S = h_\mu$.

Once the multi-collision is constructed, we carry out the generic attack in question, starting from h_S, i.e., as if h_S was the initial value of the hash function. When the generic attack is carried out, a checksum or an intermediate checksum value is fixed by the attack. From this (intermediate) checksum, the corresponding checksum output $v_S = v_\mu$ required from the multi-collision can be found. This may require a relatively small number of inversions or bridgings of the checksum function.

A message from the multi-collision producing the intermediate checksum v_S must now be found. This is done by a meet-in-the-middle attack (see Section 4.1.4). We note that this meet-in-the-middle attack requires computing the intermediate checksums of all the messages that may be produced from the first $\mu/2$ blocks in the multi-collision, and similarly, the checksum function must be inverted through all the messages that can be produced from the last $\mu/2$ blocks in the multi-collision. This would seem to have complexity $\mu 2^{\mu/2+1}$, but by exploiting the binary tree structure of the messages in the multi-collision, the complexity may be reduced to $2^{\mu/2+2}$.

The total complexity added by the checksum construction compared to the plain Merkle-Damgård construction is therefore about $\mu 2^{n/2} + 2^{\mu/2+2}$. If this complexity is below the complexity of the generic attack, then the generic attack is hardly affected by the addition of the invertible checksum. On the other hand, it may be that $\mu \geq 2n$, in which case the brute force generic attack is more efficient. We note that for most hash functions in the MD4 family, $\mu \geq 2n$, and hence the addition of a checksum might protect these. However, many hash functions (e.g., MD2) have, for instance, $\mu = n$, and in this case the checksum only adds a complexity corresponding to a collision attack.

For multi-collision attacks, a much faster technique than the one described above may be applied. The following collision attack shows why.

1: Choose an arbitrary checksum value v_{T}
2: **for** $i = 1$ to $2^{n/2}$ **do**
3: Choose a random message block m_1^i
4: Compute $v^i \leftarrow c(v_0, m_1^i)$
5: Compute m_2^i such that $c(v^i, m_2^i) = v_{\mathrm{T}}$ (by bridging)
6: **end for**{We now have $2^{n/2}$ message $m_1^i \| m_2^i$ such that $C(v_0, m_1^i \| m_2^i) = v_{\mathrm{T}}$}
7: Find i and j such that $F(\mathrm{iv}, m_1^i \| m_2^i) = F(\mathrm{iv}, m_1^j \| m_2^j)$
8: **return** $(m_1^i \| m_2^i, m_1^j \| m_2^j)$

The above attack produces a message pair $(m_1^i \| m_2^i, m_1^j \| m_2^j)$ such that

$C(v_0, m_1^i \| m_2^i) = C(v_0, m_1^j \| m_2^j)$ and $H(m_1^i \| m_2^i) = H(m_1^j \| m_2^j)$. It requires time about $2^{n/2+2}$, assuming that c can be evaluated (and bridged) in time equivalent to evaluating f. Note that the complexity of the attack is independent of μ, and therefore it applies to, e.g., the MD4 family of hash functions (having $\mu \geq 512$) when extended with an invertible checksum function. A multi-collision attack may (by a simple extension) be based on this collision attack, allowing a 2^t-collision to be constructed in time about $t2^{n/2+2}$.

We note the attack does not constitute an efficient collision attack on the entire hash function, since the birthday attack has complexity $2^{n/2}$.

4.4.2 One-way checksum function

We now assume that the checksum function is one-way, or, more generally, that inverting (or bridging) the checksum function takes time $T = 2^u$.

In this case, the initial multi-collision (the checksum control sequence) may be constructed as above, but searching for the right message in the end takes more time. The middle in the meet-in-the-middle attack is now placed at the $(\mu + u)/2$-th block, meaning that the checksum function must be evaluated $2^{(\mu+u)/2+1}$ times, and inverted $2^{(\mu-u)/2+1}$ times. The complexity is therefore about $2^{(\mu+u)/2+2}$.

Note that u can never be greater than μ, since inverting (or bridging) c by brute force takes time 2^μ. If $(\mu + u) < 2n$, then the generic attack may still be faster than a brute force attack.

As an example, consider SHA-256 "strengthened" with a 128-bit checksum computed by the MD5 hash function. The output of MD5 is padded to 512 bits, and used as a final message block in SHA-256. This checksum function can be inverted in time 2^{128}, and hence, for example, the second preimage attack of Section 2.2.4, assuming a 2^{55}-block first preimage, can be carried out in time about 2^{201}. Although being one-way, the checksum function adds no protection in this case, because it is (much) shorter than the values in the hash chain.

As above for the invertible checksum function, multi-collisions can be carried out more efficiently. Since a checksum-based hash function may be viewed as a cascaded construction [160] followed by a "merging" of the two chains, Joux's collision attack [90] on the cascaded construction applies to checksum-based hash functions. Such a collision can, again, be used to construct multi-collisions. The method to find a collision is the following.

1: Construct a $2^{n/2}$-collision on the checksum chain.
2: Among the $2^{n/2}$ messages in the multi-collision, find two messages that collide in the hash chain.

The first step takes time about $(n/2)2^{\mu/2}$. The second step requires computing $2^{n/2}$ intermediate hash values of $(n/2)$-block messages, but these form a tree structure, meaning they can be computed in time $2^{n/2+1}$. The total time complexity is therefore $(n/2)2^{\mu/2} + 2^{n/2+1}$. Note that we made no assumptions on the checksum function, except that it is assumed to require about the same time as f to evaluate.

As above, a 2^t-collision may be constructed from the above collision attack in time $t((n/2)2^{\mu/2} + 2^{n/2+1})$.

The roles of the checksum function and the compression function may be swapped. For instance, if $\mu > n$, then a slightly faster attack is obtained by first finding a $2^{\mu/2}$-collision on the hash chain, and then finding a collision in the checksum chain. The complexity is then $t(2^{\mu/2+1} + (\mu/2)2^{n/2})$.

Returning to our example of SHA-256 extended with a checksum computed via MD5, a 2^t-collision can be found in time about $t2^{129}$, compared to about $t2^{128}$ for SHA-256 without a checksum. For a one-way checksum function to offer significant protection against the multi-collision attack, the checksum must be at least twice as large as the chaining values.

4.4.3 Application to MD2

MD2 operates with n-bit message blocks ($n = 128$), and maintains an n-bit checksum state as well (see also Section 3.4.1). Hence, $\mu = n = 128$. The checksum function is non-linear, but it can be evaluated, inverted and bridged in time about 1/52 of the time required to compute f.

This means that the attacks described in Section 4.4.1 above apply to MD2. Constructing the initial multi-collision takes time about $128 \cdot 2^{64} = 2^{71}$. Searching the multi-collision at the end takes time about 2^{65}.

Hence, the second preimage attack of Kelsey and Schneier (see Section 2.2.4), given a k-block first preimage, takes time about $2^{71} + 2^{128-k}$. Note that this is no faster than the shortcut preimage attack of Section 4.2.3.

The Nostradamus attack (Section 2.2.5) may be carried out with the extension described here in time about $2^{85.3}$ (in this case, the checksum does not add to the complexity of the attack). Again, this is less efficient than the shortcut preimage attack of Section 4.2.3.

Multi-collisions can be constructed by the method described above when the checksum function is invertible. Hence, a 2^t-collision can be constructed in time about $t2^{65}$ (recall that the checksum function can be inverted much faster than the time required to evaluate f). Again, we see that the checksum adds almost no protection.

4.4.4 Summary

The discussion above means that if one wants to protect the Merkle-Damgård construction against multi-collisions, then adding a checksum is probably not the best way to do this; it would require a checksum that is at least twice as large as the hash itself, and it would need the checksum function to be one-way. It seems better to introduce some mixing between "checksum bits" and "hash bits". This solution resembles the double-pipe scheme of Lucks, see [121] and Section 3.3.2, and it can hardly be called a checksum.

For protection against shortcut attacks, a checksum may still be a good, quick solution. Recall, for instance, that the shortcut collision attack described on MD2 in Section 4.2.2 is only slightly faster than a birthday attack, whereas an efficient collision attack on MD2 without a checksum was described already in 1997 [179]. However, a simple linear checksum of message blocks or intermediate hash values is by no means guaranteed to protect, say, SHA-1 against shortcut collision attacks; what is needed to construct a collision if the checksum is, e.g., the XOR of all intermediate hash states, is simply a three-block collision attack such that the intermediate hash states after the first and the second block contain the same difference. In the currently best known collision attacks on SHA-1 [51, 52, 215], collisions of this or a similar kind may be produced.

4.5 A concrete collision attack on some rate 1/2 permutation-based hash functions

Permutation-based hash functions were introduced in Section 3.2. In this section, we describe a collision attack on a large class of $(2, 1, 2)$-constructions. The attack finds collisions in the compression function, which can sometimes be extended to the full hash function. We use the terminology of Section 3.2 in the following.

Consider a $(2, 1, 2)$-construction with g_1, g_2, and G being linear over $\mathbb{F}_{2^n}$ (the attack below also applies, with minor changes, if the three functions are affine):

$$\begin{aligned} y_1 &\leftarrow p_1(g_1(h, m)) = p_1(\mathbf{a} \cdot h + \mathbf{b} \cdot m) \\ y_2 &\leftarrow p_2(g_2(h, m, y_1)) = p_2(\mathbf{c} \cdot h + \mathbf{d} \cdot m + \mathbf{e} \cdot y_1) \\ f(h, m) &\leftarrow G(h, m, y_1, y_2) = \mathbf{f} \cdot h + \mathbf{g} \cdot m + \mathbf{h} \cdot y_1 + \mathbf{i} \cdot y_2. \end{aligned} \tag{4.5}$$

Symbols in **bold face** are constants in $\mathbb{F}_{2^n}$. See also Figure 4.17.

This construction, according to [177], can obtain collision resistance of no more than $2^{n/4}$. Here, we provide a collision attack on the compression

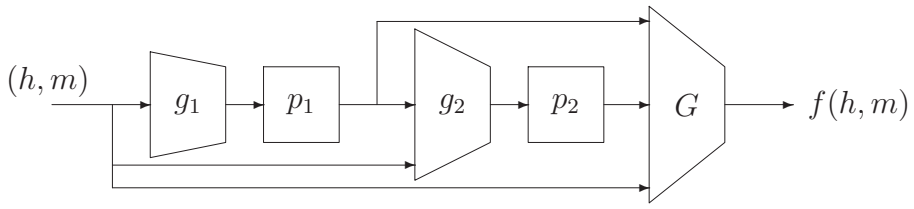


Figure 4.17: The compression function f based on two permutation calls.

function of complexity $2^{n/3}$. The attack is not query-based, but based on Wagner's generalised birthday attack (see Section 4.1.5).

If a compression function can be written as $f(h, m) = f_1(h) \oplus f_2(m)$, then Wagner's generalised birthday attack can be used to find collisions for the compression function in time $2^{n/3}$. The attack can also be applied if $f(h, m) = f_1(\alpha) \oplus f_2(\beta)$, where α and β are computed from h and m in an invertible fashion. Here, we show that this is the case for the construction (4.5). We note that the additions in the construction are in fact XORs, because they take place in $\mathbb{F}_{2^n}$.

First, we expand $f(h, m)$:

$$f(h,m) = \mathbf{f}\cdot h + \mathbf{g}\cdot m + \mathbf{h}\cdot p_1(\mathbf{a}\cdot h + \mathbf{b}\cdot m) + \mathbf{i}\cdot p_2(\mathbf{c}\cdot h + \mathbf{d}\cdot m + \mathbf{e}\cdot p_1(\mathbf{a}\cdot h + \mathbf{b}\cdot m)).$$

Let $\alpha = \mathbf{a} \cdot h + \mathbf{b} \cdot m$ and let $\beta = \mathbf{c} \cdot h + \mathbf{d} \cdot m + \mathbf{e} \cdot p_1(\alpha)$. Then we have

$$f(h, m) = \mathbf{f} \cdot h + \mathbf{g} \cdot m + \mathbf{h} \cdot p_1(\alpha) + \mathbf{i} \cdot p_2(\beta).$$

Consider the sum $(\mathbf{x} \cdot \alpha + \mathbf{y} \cdot p_1(\alpha)) + (\mathbf{z} \cdot \beta + \mathbf{i} \cdot p_2(\beta))$. This sum expands into

$$(\mathbf{ax} \cdot h + \mathbf{bx} \cdot m + \mathbf{y} \cdot p_1(\alpha)) + (\mathbf{cz} \cdot h + \mathbf{dz} \cdot m + \mathbf{ez} \cdot p_1(\alpha) + \mathbf{i} \cdot p_2(\beta)).$$

Collecting terms, one obtains

$$(\mathbf{ax} + \mathbf{cz}) \cdot h + (\mathbf{bx} + \mathbf{dz}) \cdot m + (\mathbf{y} + \mathbf{ez}) \cdot p_1(\alpha) + \mathbf{i} \cdot p_2(\beta).$$

In order to equate this expression with $f(h, m)$ we need to solve (for $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$) the following three simultaneous equations:

$$\begin{aligned} \mathbf{ax} + \mathbf{cz} &= \mathbf{f} \\ \mathbf{bx} + \mathbf{dz} &= \mathbf{g} \\ \mathbf{y} + \mathbf{ez} &= \mathbf{h}, \end{aligned}$$

which can be written as

$$\begin{pmatrix} \mathbf{a} & \mathbf{0} & \mathbf{c} \\ \mathbf{b} & \mathbf{0} & \mathbf{d} \\ \mathbf{0} & \mathbf{1} & \mathbf{e} \end{pmatrix} \cdot \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \\ \mathbf{z} \end{pmatrix} = \begin{pmatrix} \mathbf{f} \\ \mathbf{g} \\ \mathbf{h} \end{pmatrix}.$$

This system of equations has a solution whenever $\mathbf{ad} \neq \mathbf{bc}$. Once $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$ have been found, the compression function f can be written as

$$f(h, m) = (\mathbf{x} \cdot \alpha + \mathbf{y} \cdot p_1(\alpha)) + (\mathbf{z} \cdot \beta + \mathbf{i} \cdot p_2(\beta)),$$

where $\alpha = \mathbf{a} \cdot h + \mathbf{b} \cdot m$ and $\beta = \mathbf{c} \cdot h + \mathbf{d} \cdot m + \mathbf{e} \cdot p_1(\alpha)$. This means that Wagner's generalised birthday attack can be used to find the quadruple $(\alpha, \alpha^*, \beta, \beta^*)$ forming a collision. From α and β one obtains h and m by solving

$$\begin{pmatrix} \mathbf{a} & \mathbf{b} \\ \mathbf{c} & \mathbf{d} \end{pmatrix} \cdot \begin{pmatrix} h \\ m \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta - \mathbf{e} \cdot p_1(\alpha) \end{pmatrix}.$$

As above, this system has a solution if $\mathbf{ad} \neq \mathbf{bc}$.

If $\mathbf{ad} = \mathbf{bc}$, then collisions can be found in constant time via preimages for f. Let $(\mathbf{c}, \mathbf{d}) = \mathbf{k} \cdot (\mathbf{a}, \mathbf{b})$ (such a $\mathbf{k}$ exists when $\mathbf{ad} = \mathbf{bc}$). Choose arbitrary α and α^*, and compute $y_1 = p_1(\alpha)$ and $y_2 = p_2(\mathbf{k} \cdot \alpha + \mathbf{e} \cdot y_1)$, and likewise for α^*. Choose arbitrary γ, and solve the system of equations

$$\begin{pmatrix} \mathbf{a} & \mathbf{b} \\ \mathbf{f} & \mathbf{g} \end{pmatrix} \cdot \begin{pmatrix} h \\ m \end{pmatrix} = \begin{pmatrix} \alpha \\ \gamma - \mathbf{h} \cdot y_1 - \mathbf{i} \cdot y_2 \end{pmatrix}.$$

The solution gives $f(h, m) = \gamma$. A similar system of equations can be set up for α^*:

$$\begin{pmatrix} \mathbf{a} & \mathbf{b} \\ \mathbf{f} & \mathbf{g} \end{pmatrix} \cdot \begin{pmatrix} h^* \\ m^* \end{pmatrix} = \begin{pmatrix} \alpha^* \\ \gamma - \mathbf{h} \cdot y_1^* - \mathbf{i} \cdot y_2^* \end{pmatrix}.$$

Again, the solution gives (h^*, m^*) such that $f(h^*, m^*) = \gamma$. Hence, a collision has been found. Both systems of equations can be solved if $\mathbf{ag} \neq \mathbf{bf}$. If $\mathbf{ag} = \mathbf{bf}$, then any (h, h^*, m, m^*) with $\mathbf{a} \cdot h + \mathbf{b} \cdot m = \mathbf{a} \cdot h^* + \mathbf{b} \cdot m^*$ forms a collision.

We remind the reader that a collision in the compression function does not necessarily extend to a collision in the hash function. For this construction, if $\mathbf{b} = \mathbf{0}$, or if $(\mathbf{d}, \mathbf{e}) = (\mathbf{0}, \mathbf{0})$, then $h = \alpha$ or $h = \beta$, respectively, and in both cases, h goes into Wagner's generalised birthday attack directly. This means that the $2^{n/3}$ values of α or β $(= h)$ can instead be computed from, say, the initial value iv of the hash function as $f(\mathrm{iv}, m_1)$ by using $2^{n/3}$ first message blocks m_1. This only increases the complexity of the attack on the compression function by a factor about two.

As an example, the compression function $f(h, m) = p_1(h + m) + p_2(h) + m$ yields a hash function which succumbs to Wagner's generalised birthday attack, but $f^*(h, m) = p_1(h + m) + p_2(m) + h$, which is the construction used in `Grøstl`, see Section 5.1, apparently does not.

Chapter 5

The SHA-3 competition

The surge of collision attacks on widely used and standardised hash functions, particularly in the year 2005, naturally led to a decrease in confidence also in hash functions that were not yet broken, but whose designs somewhat resembled the designs of broken hash functions. The National Institute of Standards and Technology (NIST) hosted two workshops on cryptographic hash functions in 2005 and 2006 in order to assess the current status of cryptographic hash functions, and to develop some ideas of how to react to the new attacks.

It was suggested that a competition similar to the one held in the context of block ciphers at the end of the 90s (the AES competition) be initiated. Some concern was raised that the community was not ready for a competition, as the theoretical background on hash functions was deemed to be too weak. On the other hand, the AES competition proved very fruitful in terms of research on block ciphers, and it was argued that similar improvements of the state of the art might be obtained from a hash function competition. In any case, the competition (called the SHA-3 competition) was initiated in November 2007, and the deadline for submission was set to be October 31, 2008. The winner is to be selected in 2012 after an elimination round and a final. A new standard, called SHA-3, is to be published shortly thereafter.

The author was part of a team submitting a candidate for the SHA-3 competition. This candidate, named `Grøstl`, will now be described.

5.1 SHA-3 candidate: `Grøstl`

The `Grøstl` hash function [71] is a permutation based hash function (see Section 3.2), for which the compression function has a security proof. As mentioned in Section 3.2, a permutation based hash function cannot obtain

ideal collision and preimage resistance with fewer than three permutation calls per compression function call. However, with two permutation calls per compression function call, a construction with a reasonable lower bound on the complexity of collision and preimage attacks is possible; this lower bound is below the ideal complexities, but with an increased size of the permutations, the resulting hash function may still be optimally collision and preimage resistant. The validity of the bounds requires that the permutations are modelled as ideal permutations, informally meaning that nothing (except for consistency with a permutation) is known about the output of the permutation call or a call to its inverse, before the call is actually made.

5.1.1 The hash function construction

`Grøstl` is a permutation based hash function whose compression function requires two permutation calls. The permutations are at least twice as large as the final output size n of the hash function, and the compression function is iterated in the Merkle-Damgård mode, with the addition of an output transformation at the end.

Let the compression function be f, and define an initial ℓ-bit value $h_0 = \text{iv}$ to be specified below. Pad the message M, and split it into blocks $m_1, \ldots, m_t$ of ℓ bits each. Then compute

$$h_i \leftarrow f(h_{i-1}, m_i) \quad \text{for } i = 1, \ldots, t.$$

The final chaining value h_t is processed by an output transformation Ω mapping ℓ-bit inputs to n-bit outputs, and the output of the hash function is $H(M) = \Omega(h_t)$.

5.1.2 The compression function construction

As mentioned, the compression function calls two permutations, which we shall call P and Q. They are of size ℓ bits each. The compression function f is defined as

$$f(h, m) = P(h \oplus m) \oplus Q(m) \oplus h.$$

We note that this construction was (apparently) first proposed and investigated in the author's master's thesis [205, Section 8.4.4] as an attempt to strengthen SMASH [98]. The security of the construction was (more formally) investigated by Fouque, Stern, and Zimmer [68]. It was proved that a hash function iterating f in the Merkle-Damgård mode is collision resistant up to $2^{\ell/4}$ permutation calls, and preimage resistant up to $2^{\ell/2}$ permutation calls. Hence, with $\ell \geq 2n$, one has optimal collision and preimage resistance,

assuming that the permutations are ideal, and that the output transformation is secure.

5.1.3 The output transformation

An output transformation Ω is applied to reduce the size of the output from ℓ to n bits. The output transformation is defined as

$$\Omega(x) = \mathsf{trunc}_n(P(x) \oplus x).$$

The construction $P(x) \oplus x$ is often seen in cryptography, and it is generally believed to be one-way and collision resistant. Given these properties, together with the fact that the input to the output transformation is not under the direct control of the attacker, assuming P is ideal, it does not seem possible to attack `Grøstl` via the output transformation.

5.1.4 `Grøstl` instances

`Grøstl` is a collection of hash functions returning from 1 up to 64 bytes, or in other words, from 8 bits up to 512 bits in 8-bit steps. The variant returning n bits is called `Grøstl-`n. All members of the collection returning up to 256 bits are constructed in the same way; only the initial values and the truncation in the end differ. Similarly, all longer variants are constructed in the same way, but differently from the shorter variants. For the shorter variants, ℓ has the value 512; for the longer variants, it has the value 1024. Hence, P and Q come in two forms each, one is a 512-bit permutation, and one is a 1024-bit permutation, and it is always the case that $\ell \geq 2n$. When it is necessary to distinguish the permutations of different sizes, we shall denote by P_ℓ and Q_ℓ the ℓ-bit variants. We now describe the permutations P and Q.

5.1.5 The permutations P and Q

The permutations are inspired by the Rijndael block cipher [42]. However, Rijndael operates with a state of only 128 bits, and P and Q are of size at least 512 bits.

The state of P and Q is seen as a matrix of bytes. For both permutation sizes, this matrix has 8 rows. For the 512-bit permutations, the number of columns is 8, and for the 1024-bit permutations, the number of columns is 16. P and Q are constructed from a number of rounds, where each round applies the following state transformations (in order):

- AddRoundConstant
- SubBytes
- ShiftBytes
- MixBytes.

The transformation AddRoundConstant replaces the Rijndael transformation called AddRoundKey. The SubBytes transformation is defined in the same way as in Rijndael. The transformations ShiftBytes and MixBytes are variants of the Rijndael transformations ShiftRows and MixColumns, respectively. Since the state (and hence matrix) size is not the same in P and Q as in Rijndael, these transformations had to be redefined. However, the purpose of the transformations is the same as their Rijndael counterparts. The ShiftBytes transformation is different for the two different permutation sizes; when we need to distinguish between them, we denote the variant used in the larger permutations by ShiftBytesWide. To summarise, a round R is defined as

$$R = \mathsf{MixBytes} \circ \mathsf{ShiftBytes} \circ \mathsf{SubBytes} \circ \mathsf{AddRoundConstant}.$$

We state the recommended number of rounds in P and Q later.

To be able to describe the four state transformations for the two different permutation sizes in general, we denote by v the number of columns in the state matrix. We denote the matrix by A, and its elements as $a_{i,j}$, meaning the element in row i, column j. Mapping a 64-byte, respectively 128-byte string to an 8×8, respectively 8×16 state matrix is done by placing byte no. i in column $\lfloor i/8 \rfloor$ and row $i \bmod 8$. In the following, this mapping back and forth between byte strings and state matrices is not mentioned explicitly.

AddRoundConstant

The AddRoundConstant transformation XORs a round-dependent constant onto the state matrix A. P and Q have different round constants, which is the only difference between the two permutations.

The round constants can be seen as matrices of the same size as the state matrix. All round constant bytes are zero except in a single position. The round constants used for P_{512} and P_{1024} are basically the same: The first 8 columns do not differ for both permutations and the last 8 columns of the round constants used in P_{1024} contain only bytes having the value 00. Likewise for Q_{512} and Q_{1024}.

The byte in the top leftmost corner of the round constant in round i of P has the value i; all other positions in the round constant matrix have the

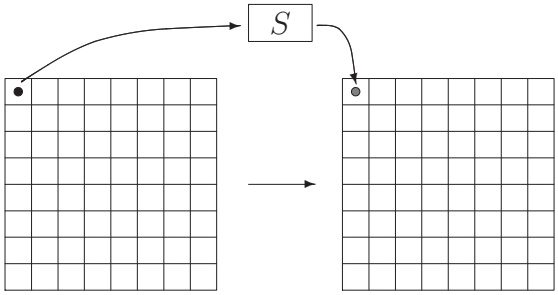


Figure 5.1: The SubBytes transformation.

value 00. In Q, the byte in the bottom leftmost corner has the value $i \oplus \mathsf{ff}$, and all other bytes have the value 00. The round number is reduced modulo 256, if necessary.

To be precise, the AddRoundConstant transformation in round i updates the state A as

$$A \leftarrow A \oplus C[i],$$

where $C[i]$ is the round constant used in round i. The round constants $C_P[i]$ and $C_Q[i]$ used in round i of P and Q, respectively, are

$$C_P[i] = \begin{bmatrix} i & 00 & \cdots & 00 \\ 00 & 00 & \cdots & 00 \\ 00 & 00 & \cdots & 00 \\ 00 & 00 & \cdots & 00 \\ 00 & 00 & \cdots & 00 \\ 00 & 00 & \cdots & 00 \\ 00 & 00 & \cdots & 00 \\ 00 & 00 & \cdots & 00 \end{bmatrix} \quad \text{and} \quad C_Q[i] = \begin{bmatrix} 00 & 00 & \cdots & 00 \\ 00 & 00 & \cdots & 00 \\ 00 & 00 & \cdots & 00 \\ 00 & 00 & \cdots & 00 \\ 00 & 00 & \cdots & 00 \\ 00 & 00 & \cdots & 00 \\ 00 & 00 & \cdots & 00 \\ i \oplus \mathsf{ff} & 00 & \cdots & 00 \end{bmatrix}.$$

SubBytes

The SubBytes transformation substitutes each byte in the state matrix by another value, taken from the S-box S. This S-box is the same as the one used in Rijndael and its specification can be found in Table 5.1. Hence, SubBytes performs the following transformation:

$$a_{i,j} \leftarrow S(a_{i,j}), \quad 0 \le i < 8,\ 0 \le j < v.$$

See also Figure 5.1.

Table 5.1: The Grøstl S-box (identical to the Rijndael/AES S-box).

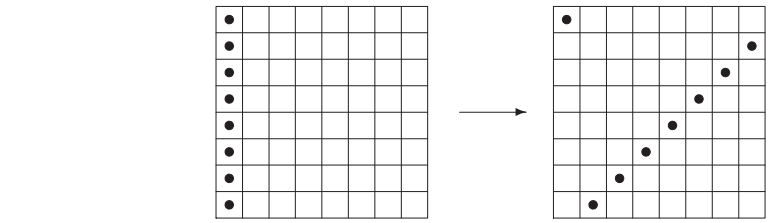
	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
10	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
20	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
30	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
40	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
50	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
60	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
70	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
80	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
90	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a0	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b0	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c0	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d0	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e0	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f0	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

ShiftBytes and ShiftBytesWide

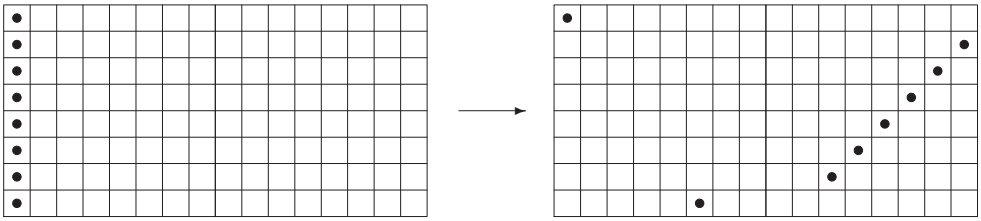
ShiftBytes and ShiftBytesWide cyclically shift the bytes within a row to the left by a number of positions. Let $\sigma = [\sigma_0, \sigma_1, \ldots, \sigma_7]$ be a list of distinct integers in the range from 0 to $v - 1$. Then, ShiftBytes moves all bytes in row i of the state matrix σ_i positions to the left, wrapping around as necessary. The vector σ is defined as $\sigma = [0, 1, 2, 3, 4, 5, 6, 7]$ in ShiftBytes, and $\sigma = [0, 1, 2, 3, 4, 5, 6, 11]$ in ShiftBytesWide. See Figure 5.2.

MixBytes

In the MixBytes transformation, each column in the matrix is transformed independently. To describe this transformation we first need to introduce the finite field $\mathbb{F}_{256}$. This finite field is defined in the same way as in Rijndael, i.e., via the irreducible polynomial $x^8 \oplus x^4 \oplus x^3 \oplus x \oplus 1$ over $\mathbb{F}_2$. The bytes of the state matrix A can be seen as elements of $\mathbb{F}_{256}$, i.e., as polynomials of degree at most 7 with coefficients in $\{0, 1\}$. The least significant bit of a byte determines the coefficient to x^0, etc.



(a) ShiftBytes



(b) ShiftBytesWide

Figure 5.2: The ShiftBytes transformations.

MixBytes multiplies each column of A by a constant 8×8 matrix B in $\mathbb{F}_{256}$. Hence, the transformation on the whole matrix A can be written as the matrix multiplication

$$A \leftarrow B \times A.$$

The matrix B is specified as

$$B = \begin{bmatrix} 02 & 02 & 03 & 04 & 05 & 03 & 05 & 07 \\ 07 & 02 & 02 & 03 & 04 & 05 & 03 & 05 \\ 05 & 07 & 02 & 02 & 03 & 04 & 05 & 03 \\ 03 & 05 & 07 & 02 & 02 & 03 & 04 & 05 \\ 05 & 03 & 05 & 07 & 02 & 02 & 03 & 04 \\ 04 & 05 & 03 & 05 & 07 & 02 & 02 & 03 \\ 03 & 04 & 05 & 03 & 05 & 07 & 02 & 02 \\ 02 & 03 & 04 & 05 & 03 & 05 & 07 & 02 \end{bmatrix}.$$

This matrix is *circulant*, which means that each row is equal to the row above rotated right by one position. In short, we may write

$$B = \mathrm{circ}(02, 02, 03, 04, 05, 03, 05, 07)$$

instead. See also Figure 5.3.

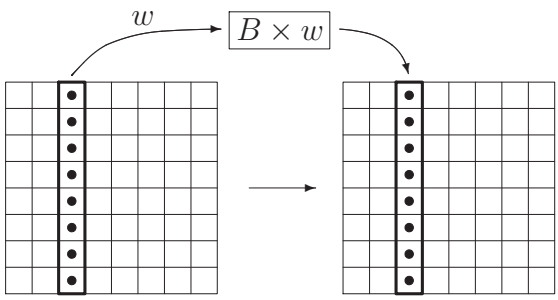


Figure 5.3: The MixBytes transformation.

Number of rounds

The number r of rounds is a tunable security parameter. We recommend the following values of r for the four permutations.

Permutations	Digest sizes	Recommended value of r
P_{512} and Q_{512}	8–256	10
P_{1024} and Q_{1024}	264–512	14

5.1.6 Padding

The input message M is padded as follows. Append a '1' bit to the message, and then append $w = -|M| - 65 \bmod \ell$ '0' bits. Finally, append a 64-bit representation of the value $(|M| + w + 65)/\ell$, which, due to the choice of w, is always an integer. This number represents the number of message blocks in the padded message.

The maximum message length is restricted by the padding method as follows. Since it must be possible to encode the number of message blocks in the padded message within 64 bits, there can be at most $2^{64} - 1$ message blocks in the padded message. Padding always appends at least 65 bits, and hence the maximum message length is $(2^{64} - 1)\ell - 65$ bits. With $\ell = 512$, i.e., for `Grøstl` variants returning up to 256 bits, the maximum message length (in bits) is therefore $512 \cdot (2^{64} - 1) - 65 = 2^{73} - 577$, and for the longer variants it is $1024 \cdot (2^{64} - 1) - 65 = 2^{74} - 1089$.

5.1.7 Initial values

The initial value iv_n of `Grøstl-`n is the ℓ-bit representation of n. The table below shows the initial values of the output sizes of 224, 256, 384, and 512 bits.

n	iv_n
224	00 ... 00 00 e0
256	00 ... 00 01 00
384	00 ... 00 01 80
512	00 ... 00 02 00

5.1.8 Grøstl features

As mentioned, the construction underlying the Grøstl compression function has been proved to be secure, assuming ideal permutations. No permutation occurring in practice can be ideal; merely the fact that there is an algorithm that describes what takes place inside the permutation disqualifies the permutation from being ideal. On the other hand, an indication that the permutation does not behave ideally does not necessarily lead to an attack on the hash function.

The compression function

Wagner's generalised birthday attack, see [213] and Section 4.1.5, applies to the Grøstl compression function: let $f_P(x) = P(x) \oplus x$, and let $f_Q(x) = Q(x) \oplus x$. Then, f can be described as $f_P(h \oplus m) \oplus f_Q(m)$, and therefore, the attack applies. However, the attack does not allow the attacker to choose h, and therefore it does not seem possible to extend the attack on the compression function to the full hash function. Moreover, the attack has complexity $2^{\ell/3}$, which is above the proven bound of $2^{\ell/4}$, and also, since $\ell \geq 2n$, above the complexity of the birthday attack on the full hash function.

Intuitively, if f_P and f_Q are modelled as random functions, then the best method to find a collision for f is to compute $2^{\ell/4}$ output values of each of f_P and f_Q, and combine them in $2^{\ell/2}$ different ways, yielding a good probability of a collision. Hence, this informally shows that the compression function f is collision resistant up to $2^{\ell/4}$ permutation calls, assuming that P and Q are ideal permutations. If there was no output transformation, then the collision resistance of the compression function would extend to the full hash function via the same proof as the security proof of the Merkle-Damgård construction. Hence, finding a collision in the hash function requires finding a collision in the compression function *or* in the output transformation.

Wagner's generalised birthday attack is the best known collision attack on the Grøstl compression function. The best known collision attack on the Grøstl hash function is the birthday attack. Similarly, the best known collision attack on the compression function with a given, fixed chaining input

is the birthday attack.

AddRoundConstant

The purpose of adding round constants is to make each round different, and at the same time this provides a natural opportunity to differentiate P and Q. If the rounds are all the same, then fixed points x such that $R(x) = x$ for the round function R extend to the entire permutation. For example, if $P = R^{10}$, then fixed points for R^2 and R^5 would also extend to P. Therefore, one can expect several fixed points for P, whereas for an ideal permutation, only a single fixed point is expected. By choosing round-dependent constants for AddRoundConstant, the number of fixed points of P and Q is expected to be 1.

Simple round constants with zeroes in most positions are used in order to reduce the performance penalty of this transformation. Since this is the only transformation in which there is a difference between P and Q, the round constants must be different for P and Q.

SubBytes

The SubBytes transformation is the only non-linear transformation in `Grøstl`. It uses the same S-box as used in Rijndael. For a walk-through of its properties, we refer to one of [42, 44].

ShiftBytes and ShiftBytesWide

The shift values used in ShiftBytes affect the speed of diffusion in P and Q. For the 512-bit permutations, every byte of the state affects every other byte after two rounds. For the 1024-bit permutations, this happens after three rounds. In both cases, the speed of diffusion is optimal for the respective state geometries.

MixBytes

The main design goal of the MixBytes transformation is to follow the wide trail strategy [43]. Hence, the MixBytes transformation is based on an error-correcting code with the MDS (maximum distance separable) property. This ensures that the branch number is 9. In other words, a difference in $k > 0$ bytes of a column will result in a difference in at least $9 - k$ bytes after one MixBytes application.

There exist many MDS codes with the required parameters. `Grøstl` uses a code which can be implemented efficiently in many settings. The

MixBytes transformation multiplies each column of A with the MDS matrix $B = \text{circ}(02, 02, 03, 04, 05, 03, 05, 07)$ over the finite field $\mathbb{F}_{256}$. In most environments, the multiplication with a constant of this matrix is the most expensive part. The implementation costs can be reduced by using constants of low degree. The minimum degree of the constants for an MDS code of size 8 is 2. However, this comes at a higher cost for the additions due to a slightly higher Hamming weight of the elements. In Grøstl, the values are chosen such that the additional costs of additions can be compensated by combining intermediate results. Particularly on 8-bit platforms, this results in more efficient implementations. We note that on modern PC platforms, the exact specification of B has no impact on the performance.

Output transformation

Since the size of the chaining variables is larger than the required output size, an output transformation is needed. Simple truncation would be a possibility. However, since the compression function is not ideal, we chose to apply a function which is believed to be one-way and collision resistant, but does not compress before the truncation. Also, as mentioned, a collision in the hash function implies a collision in the compression function or in the output transformation, and therefore it makes sense to strengthen the output transformation so that it is believed to be collision resistant.

Let $\omega(x) = P(x) \oplus x$. The Matyas-Meyer-Oseas construction (Construction 3.2) for hash functions based on block ciphers provides a compression function g based on the encryption function E_K (with K being the key) as follows:

$$g(h, m) = E_h(m) \oplus m.$$

This function g has been proved to provide a collision resistant and one-way hash function when iterated in the Merkle-Damgård mode [24], under the assumption that E is an ideal block cipher. This implies that g is collision resistant and one-way if h is fixed, since this corresponds to hashing a one-block message. Hence, $\tilde{g}(m) = E_{h^*}(m) \oplus m$, where h^* is a constant, is one-way and collision resistant as well. Since $\tilde{g} = \omega$ with $P = E_{h^*}$, ω may reasonably be believed to be one-way and collision resistant. This seems to make it difficult to attack Grøstl via the output transformation.

5.1.9 Preliminary cryptanalysis results

In this section, we describe the results of a preliminary cryptanalytic investigation of Grøstl.

Generic attacks

With its large internal state size, `Grøstl` seems to protect against many of the generic attacks that apply to Merkle-Damgård hash functions (see Section 2.2).

With the best known collision attack on the compression function f with fixed chaining input being the birthday attack, the multi-collision attack of Joux (Section 2.2.3) on `Grøstl` is less efficient than a brute force multi-collision attack; it requires time $t2^{\ell/2}$ to find a 2^t-collision, and with $\ell \geq 2n$, the complexity is at least $t2^n$.

Similarly, the linking part of the second preimage attack of Kelsey and Schneier (Section 2.2.4) takes time about $2^{\ell-k}$ for a 2^k-block first preimage, and with $k \leq 64$, the complexity is always above 2^n. We do note, however, that fixed points can be used to produce the expandable message in the case of `Grøstl`. The complexity of finding a fixed point is equivalent to one compression function evaluation (choose m and compute $h = P^{-1}(Q(m)) \oplus m$), but constructing an expandable message using fixed points is expected to take time $2^{\ell/2} \geq 2^n$.

With respect to the length extension attack (Section 2.2.2) that enables the attacker to form many collisions from a single, initial collision, this attack can be applied to `Grøstl` if the initial collision occurs before the output transformation (i.e., is an internal collision). However, finding an internal collision, as mentioned, takes an expected time $2^{\ell/2} \geq 2^n$. Initial collisions that occur in the output transformation do not lead to other collisions (we note that a random collision most likely occurs in the output transformation). The output transformation also protects against the length extension attack which allows to compute a hash value without knowing the message.

To summarise, generic attacks that apply to the Merkle-Damgård construction do not apply to `Grøstl` directly via birthday and brute force methods.

Classical attacks

The effectiveness of classical attacks such as differential [18, 19], linear [125, 126], integral [41, 108] and algebraic [38, 96] attacks on Rijndael have already been studied quite extensively, with no significant success. This research may also be applied to `Grøstl`, and although an attacker has more degrees of freedom when attacking a hash function than he has when attacking a block cipher, these classical attacks do not seem to apply to `Grøstl`. For instance, the lowest number of active bytes in a four-round differential trail is 81 for both permutation sizes [44, Theorem 9.5.1]. In an eight-round

trail, the lowest number of active bytes is 162, and in a twelve-round trail it is 243. These numbers, combined with the maximum difference propagation probability of the S-box of 2^{-6}, mean that the probabilities of any differential trail (assuming independent rounds) over eight and twelve rounds (for either P or Q) are expected to be at most $2^{-6 \cdot 162} = 2^{-972}$, respectively 2^{-1458}.

5.1.10 Grøstl implementations

In this section, we briefly describe the implementability of Grøstl.

In software, Grøstl performs well. Grøstl implementations particularly take advantage of 64-bit instructions, and hence, the performance of Grøstl is particularly good on 64-bit processors, and on processors offering "vector instructions" such as those of the MMX, SSE, and SSE2 instruction sets. Almost all 32-bit Intel and AMD processors since 1997 offer this type of instructions. The 64-bit instructions, if available, are applied to one column at the time, meaning that each new column in a round of P and Q can be computed effectively using 8 table look-ups and 7 XORs.

As an example, Grøstl-256 (and shorter variants) operates at around 25 cycles/byte on a 64-bit processor such as the Intel Core 2 Duo. Longer variants perform at around 37 cycles/byte. Additional benchmarks may be found in [64, 71].

New AES instructions offered by Intel on future processors [85] may be used to develop Grøstl implementations that are both efficient and resistant to side-channel attacks such as cache-timing attacks.

Since P and Q can be described as consisting of operations on bytes, performance on 8-bit processors is also good. Moreover, many trade-offs between running time and memory usage are possible.

Similarly, several trade-offs are also possible when it comes to hardware implementations; for more information, see [71].

5.1.11 Summary

Grøstl is a permutation based hash function. The permutations were developed with the block cipher Rijndael as the main source of inspiration. Grøstl operates with a large internal state size, and the compression function construction comes with a security proof. The fate of Grøstl in the SHA-3 competition may be followed via the Grøstl web site [78].

5.2 Other SHA-3 candidates

Sixty-four candidates were submitted to the SHA-3 competition. Within the first week, four out of around 20 published candidates were broken in the sense of attacks that were carried out in practice (`Grøstl` was not among these four). One candidate, Vortex [79], is based upon the MDC-2 construction method (Construction 3.5), but with a different mixing function after the block cipher applications. In Section 4.3.5 we (briefly) described how to find collisions and preimages in Vortex.

As of November 26, 2008, 31 SHA-3 candidates were publicly known, according to the "SHA-3 Zoo" [194]. NIST had not yet finished its formal checks of all 64 submissions, and most likely, some of these will not be accepted as "complete and proper", and therefore will not enter the competition. Those that are accepted, will be published by NIST when all submissions have been checked.

Chapter 6

Conclusions

The topic of cryptographic hash functions has received an enormous amount of attention in the cryptographic community in the last few years, and with the initiation of a cryptographic hash function competition, the attention will only increase. The enthusiasm is settled on the rather disturbing fact that hash functions in wide use can no longer be considered secure.

Hash functions are expected to be extremely efficient. The question is whether it is possible to construct a secure hash function that is faster than the fastest, secure block ciphers. The fundamental difference between these two types of cryptographic primitives is that a block cipher accepts a key input, whereas as a traditional cryptographic hash function does not. Hence, in a cryptographic hash function, all information is public, which gives an attacker a huge amount of freedom when carrying out attacks. The attacker can choose to hash any message, and thereby compute the same intermediate and resulting hash values as any legitimate user of the hash function. The attacker can pre-compute values, and use them any time in the future. The attacker can carry out a collision attack with no input. Moreover, an n-bit block cipher offers up to 2^n security, but an n-bit hash function offers at most $2^{n/2}$ security. Therefore, a hash function needs to operate with a larger state than a block cipher. These issues, along with practical experience, indicate that designing a secure and efficient cryptographic hash function may be more challenging than designing a secure and efficient block cipher.

Another question one might ask is whether it makes sense to use one, single cryptographic hash function in so many different applications, or whether we should rather go back to separating hash functions into sub-classes such as one-way hash functions, collision resistant hash functions, checksum functions etc. An argument that one sometimes comes across for the high efficiency needed of a cryptographic hash function is, that someone might want to compute the checksum of a hard drive, possibly several terabytes in size.

To give a feeling for the complexity of this task, it takes at least an hour to encrypt this amount of data using the AES on a standard (year 2008) PC. The most efficient cryptographic hash functions can do it in about half an hour. If one wants to be able to do it in a minute or two, one should not use a cryptographic primitive. This illustrates that the amount of security that one requires needs to be compared against the amount of data that must be processed. Extremely high security does not go well with extremely large amounts of data – unless one is prepared to wait for an extremely long time.

A third question one might ask is whether preimage and second preimage resistance at the optimal levels of 2^n are required, or whether the birthday bound of $2^{n/2}$ is good enough for all kinds of attack. We have seen quite a large number of examples of attacks of a generic nature, that have complexities somewhere in between the birthday bound and the "brute force bound". Examples are the multi-collision attack by Joux, the second preimage attack by Kelsey and Schneier, the Nostradamus attack by Kelsey and Kohno, and the attacks on checksum-based hash functions. It is likely that new applications, variants, and extensions of these attacks will appear in the future. It is important to note, that all these mentioned attacks require the ability to produce a collision of some form or another. This fact, combined with the fact that hash functions are nearly always expected to be collision resistant in the first place, thus placing restrictions on the output size, could be interpreted as being evidence that the birthday bound should be appropriate and sufficient for all types of attack. An argument that goes against this view is that a collision found in the future will have little impact on documents that are prepared today, whereas a preimage or a second preimage may be devastating. Hence, preimage resistance and second preimage resistance are, so to speak, required to reach further into the future than collision resistance.

The SHA-3 competition will teach the cryptographic community a lot about the design and cryptanalysis of hash functions. The candidates have very different properties, and they will be attacked in many different ways. Some candidates are extremely fast, and others are slower than the SHA-2 hash functions. In the call for submissions to the SHA-3 competition, NIST requires that the winner be at least as secure, and significantly faster than SHA-2. Therefore, it is likely that the candidate that is chosen in the end will be one of the fastest candidates, for which no weaknesses are found during the competition. It will be interesting to see whether the winner turns out to be as good a choice as Rijndael (so far) turned out to be for the Advanced Encryption Standard. It will also be interesting to see just how fast the winner will be.

Bibliography

[1] C. M. Adams, G. Kramer, S. Mister, and R. J. Zuccherato. On The Security of Key Derivation Functions. In K. Zhang and Y. Zheng, editors, *International Conference on Information Security (ISC) 2004, Proceedings*, volume 3225 of *Lecture Notes in Computer Science*, pages 134–145. Springer, 2004.

[2] American National Standards Institution. ANSI X9.71. Keyed Hash Message Authentication Code, 2000.

[3] R. J. Anderson, E. Biham, and L. R. Knudsen. Serpent: A Proposal for the Advanced Encryption Standard. AES Algorithm Submission, 1998. Available: `http://www.cl.cam.ac.uk/~rja14/Papers/serpent.pdf` (2008/11/06).

[4] E. Andreeva, G. Neven, B. Preneel, and T. Shrimpton. Three-property preserving iterations of keyless compression functions. Presented at *ECRYPT Hash Workshop, May 24–25, 2007, Barcelona, Spain.* Available: `http://events.iaik.tugraz.at/HashWorkshop07/papers/Andreeva_Three-propertyPreservingIterations.pdf` (2008/09/18).

[5] E. Andreeva, G. Neven, B. Preneel, and T. Shrimpton. Seven-Property-Preserving Iterated Hashing: ROX. In K. Kurosawa, editor, *Advances in Cryptology – ASIACRYPT 2007, Proceedings*, volume 4833 of *Lecture Notes in Computer Science*, pages 130–146. Springer, 2007.

[6] P. S. L. M. Barreto and V. Rijmen. The Whirlpool Hashing Function. Submitted to NESSIE, September 2000. Revised May 2003. Available: `http://paginas.terra.com.br/informatica/paulobarreto/WhirlpoolPage.html` (2008/07/08).

[7] M. Bellare, R. Canetti, and H. Krawczyk. Keying Hash Functions for Message Authentication. In N. Koblitz, editor, *Advances in Cryptology*

– *CRYPTO '96, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1996.

[8] M. Bellare and T. Ristenpart. Multi-Property-Preserving Hash Domain Extension and the EMD Transform. In X. Lai and K. Chen, editors, *Advances in Cryptology – ASIACRYPT 2006, Proceedings*, volume 4284 of *Lecture Notes in Computer Science*, pages 299–314. Springer, 2006.

[9] M. Bellare and P. Rogaway. Random Oracles are Practical: A Paradigm for Designing Efficient Protocols. In *First ACM Conference on Computer and Communications Security, Proceedings*, pages 62–73, 1993.

[10] D. J. Bernstein. Personal communication, January 2008.

[11] T. A. Berson. Differential Cryptanalysis Mod 2^{32} with Applications to MD5. In R. A. Rueppel, editor, *Advances in Cryptology – EUROCRYPT '92, Proceedings*, volume 658 of *Lecture Notes in Computer Science*, pages 71–80. Springer, 1993.

[12] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. RadioGatún, a Belt-and-Mill Hash Function. Presented at *Second NIST Cryptographic Hash Workshop, August 24–25, 2006, Santa Barbara, California, USA.* Available: `http://radiogatun.noekeon.org/RadioGatun.pdf` (2008/11/25).

[13] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Sponge Functions. Presented at *ECRYPT Hash Workshop, May 24–25, 2007, Barcelona, Spain.* Available: `http://sponge.noekeon.org/SpongeFunctions.pdf` (2008/10/01).

[14] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. On the Indifferentiability of the Sponge Construction. In N. Smart, editor, *Advances in Cryptology – EUROCRYPT 2008, Proceedings*, volume 4965 of *Lecture Notes in Computer Science*, pages 181–197. Springer, 2008.

[15] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Keccak specifications. SHA-3 Algorithm Submission, October 2008. Available: `http://keccak.noekeon.org/Keccak-specifications.pdf` (2008/11/07).

[16] E. Biham and R. Chen. Near-Collisions of SHA-0. In M. K. Franklin, editor, *Advances in Cryptology – CRYPTO 2004, Proceedings*, volume 3152 of *Lecture Notes in Computer Science*, pages 290–305. Springer, 2004.

[17] E. Biham, R. Chen, A. Joux, P. Carribault, C. Lemuet, and W. Jalby. Collisions of SHA-0 and Reduced SHA-1. In R. Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005, Proceedings*, volume 3494 of *Lecture Notes in Computer Science*, pages 36–57. Springer, 2005.

[18] E. Biham and A. Shamir. Differential Cryptanalysis of DES-like Cryptosystems. In A. Menezes and S. A. Vanstone, editors, *Advances in Cryptology – CRYPTO '90, Proceedings*, volume 537 of *Lecture Notes in Computer Science*, pages 2–21. Springer, 1991.

[19] E. Biham and A. Shamir. Differential Cryptanalysis of DES-like Cryptosystems. *Journal of Cryptology*, 4(1):3–72, 1991.

[20] A. Biryukov. The Design of a Stream Cipher LEX. In E. Biham and A. M. Youssef, editors, *Selected Areas in Cryptography 2006, Proceedings*, volume 4356 of *Lecture Notes in Computer Science*, pages 67–75. Springer, 2007.

[21] J. Black. The Ideal-Cipher Model, Revisited: An Uninstantiable Blockcipher-Based Hash Function. In M. J. B. Robshaw, editor, *Fast Software Encryption 2006, Proceedings*, volume 4047 of *Lecture Notes in Computer Science*, pages 328–340. Springer, 2006.

[22] J. Black, M. Cochran, and T. Highland. A Study of the MD5 Attacks: Insights and Improvements. In M. J. B. Robshaw, editor, *Fast Software Encryption 2006, Proceedings*, volume 4047 of *Lecture Notes in Computer Science*, pages 262–277. Springer, 2006.

[23] J. Black, M. Cochran, and T. Shrimpton. On the Impossibility of Highly-Efficient Blockcipher-Based Hash Functions. In R. Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005, Proceedings*, volume 3494 of *Lecture Notes in Computer Science*, pages 526–541. Springer, 2005.

[24] J. Black, P. Rogaway, and T. Shrimpton. Black-Box Analysis of the Block-Cipher-Based Hash-Function Constructions from PGV. In M. Yung, editor, *Advances in Cryptology – CRYPTO 2002, Proceedings*, volume 2442 of *Lecture Notes in Computer Science*, pages 320–335. Springer, 2002.

[25] D. Boneh and M. K. Franklin. Efficient Generation of Shared RSA Keys (Extended Abstract). In B. S. Kaliski Jr., editor, *Advances in Cryptology – CRYPTO '97, Proceedings*, volume 1294 of *Lecture Notes in Computer Science*, pages 425–439. Springer, 1997.

[26] B. O. Brachtl, D. Coppersmith, M. M. Hyden, S. M. Matyas, Jr., C. H. W. Meyer, J. Oseas, S. Pilpel, and M. Schilling. Data authentication using modification detection codes based on a public one way encryption function, March 13, 1990. US Patent no. 4,908,861. Assigned to IBM. Filed August 28, 1987. Available: `http://www.google.com/patents?vid=USPAT4908861` (2008/09/02).

[27] R. P. Brent. An improved Monte Carlo factorization algorithm. *BIT*, 20(2):176–184, 1980.

[28] L. Brown, J. Pieprzyk, and J. Seberry. LOKI - A Cryptographic Primitive for Authentication and Secrecy Applications. In J. Seberry and J. Pieprzyk, editors, *Advances in Cryptology – AUSCRYPT '90, Proceedings*, volume 453 of *Lecture Notes in Computer Science*, pages 229–236. Springer, 1990.

[29] R. Canetti, O. Goldreich, and S. Halevi. The Random Oracle Methodology, Revisited (Preliminary Version). In *30th ACM Symposium on the Theory of Computing 1998, Proceedings*, pages 209–218. ACM, 1998.

[30] C. D. Cannière and C. Rechberger. Preimages for Reduced SHA-0 and SHA-1. In D. Wagner, editor, *Advances in Cryptology – CRYPTO 2008, Proceedings*, volume 5157 of *Lecture Notes in Computer Science*, pages 179–202. Springer, 2008.

[31] F. Chabaud and A. Joux. Differential Collisions in SHA-0. In H. Krawczyk, editor, *Advances in Cryptology – CRYPTO '98, Proceedings*, volume 1462 of *Lecture Notes in Computer Science*, pages 56–71. Springer, 1998.

[32] D. Chang, S. Lee, M. Nandi, and M. Yung. Indifferentiable Security Analysis of Popular Hash Functions with Prefix-Free Padding. In X. Lai and K. Chen, editors, *Advances in Cryptology – ASIACRYPT 2006, Proceedings*, volume 4284 of *Lecture Notes in Computer Science*, pages 283–298. Springer, 2006.

[33] D. Chaum, E. van Heijst, and B. Pfitzmann. Cryptographically Strong Undeniable Signatures, Unconditionally Secure for the Signer. In J. Feigenbaum, editor, *Advances in Cryptology – CRYPTO '91, Proceedings*, volume 576 of *Lecture Notes in Computer Science*, pages 470–484. Springer, 1992.

[34] S. Contini, A. K. Lenstra, and R. Steinfeld. VSH, an Efficient and Provable Collision-Resistant Hash Function. In S. Vaudenay, editor,

Advances in Cryptology – EUROCRYPT 2006, Proceedings, volume 4004 of *Lecture Notes in Computer Science*, pages 165–182. Springer, 2006.

[35] D. Coppersmith. Another Birthday Attack. In H. C. Williams, editor, *Advances in Cryptology – CRYPTO '85, Proceedings*, volume 218 of *Lecture Notes in Computer Science*, pages 14–17. Springer, 1986.

[36] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[37] J.-S. Coron, Y. Dodis, C. Malinaud, and P. Puniya. Merkle-Damgård Revisited: How to Construct a Hash Function. In V. Shoup, editor, *Advances in Cryptology – CRYPTO 2005, Proceedings*, volume 3621 of *Lecture Notes in Computer Science*, pages 430–448. Springer, 2005.

[38] N. Courtois, A. Klimov, J. Patarin, and A. Shamir. Efficient Algorithms for Solving Overdefined Systems of Multivariate Polynomial Equations. In B. Preneel, editor, *Advances in Cryptology – EUROCRYPT 2000, Proceedings*, volume 1807 of *Lecture Notes in Computer Science*, pages 392–407. Springer, 2000.

[39] J. Daemen. *Cipher and hash function design strategies based on linear and differential cryptanalysis*. PhD thesis, Katholieke Universiteit Leuven, March 1995.

[40] J. Daemen and C. S. K. Clapp. Fast Hashing and Stream Encryption with Panama. In S. Vaudenay, editor, *Fast Software Encryption '98, Proceedings*, volume 1372 of *Lecture Notes in Computer Science*, pages 60–74. Springer, 1998.

[41] J. Daemen, L. R. Knudsen, and V. Rijmen. The Block Cipher Square. In E. Biham, editor, *Fast Software Encryption 1997, Proceedings*, volume 1267 of *Lecture Notes in Computer Science*, pages 149–165. Springer, 1997.

[42] J. Daemen and V. Rijmen. AES Proposal: Rijndael. AES Algorithm Submission, September 3, 1999. Available: `http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf` (2008/10/29).

[43] J. Daemen and V. Rijmen. The Wide Trail Design Strategy. In B. Honary, editor, *Cryptography and Coding 2001, Proceedings*, volume 2260 of *Lecture Notes in Computer Science*, pages 222–238. Springer, 2001.

[44] J. Daemen and V. Rijmen. *The Design of Rijndael*. Springer, 2002.

[45] J. Daemen and V. Rijmen. A New MAC Construction ALRED and a Specific Instance ALPHA-MAC. In H. Gilbert and H. Handschuh, editors, *Fast Software Encryption 2005, Proceedings*, volume 3557 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2005.

[46] W. Dai. Crypto++® Library 5.5.2 (2008). `http://www.cryptopp.com` (2008/09/30).

[47] I. Damgård. Collision Free Hash Functions and Public Key Signature Schemes. In D. Chaum and W. L. Price, editors, *Advances in Cryptology – EUROCRYPT '87, Proceedings*, volume 304 of *Lecture Notes in Computer Science*, pages 203–216. Springer, 1988.

[48] I. Damgård. A Design Principle for Hash Functions. In G. Brassard, editor, *Advances in Cryptology – CRYPTO '89, Proceedings*, volume 435 of *Lecture Notes in Computer Science*, pages 416–427. Springer, 1990.

[49] I. B. Damgård, L. R. Knudsen, and S. S. Thomsen. Dakota – Hashing from a Combination of Modular Arithmetic and Symmetric Cryptography. In S. M. Bellovin, R. Gennaro, A. D. Keromytis, and M. Yung, editors, *Applied Cryptography and Network Security (ACNS) 2008, Proceedings*, volume 5037 of *Lecture Notes in Computer Science*, pages 144–155. Springer, 2008.

[50] D. W. Davies and W. L. Price. The Application of Digital Signatures based on Public Key Cryptosystems. Technical Report DNACS 39/80, National Physical Lab, Teddington, Middlesex, England, 1980.

[51] C. De Cannière, F. Mendel, and C. Rechberger. Collisions for 70-Step SHA-1: On the Full Cost of Collision Search. In C. M. Adams, A. Miri, and M. J. Wiener, editors, *Selected Areas in Cryptography 2007, Proceedings*, volume 4876 of *Lecture Notes in Computer Science*, pages 56–73. Springer, 2007.

[52] C. De Cannière and C. Rechberger. Finding SHA-1 Characteristics: General Results and Applications. In X. Lai and K. Chen, editors, *Advances in Cryptology – ASIACRYPT 2006, Proceedings*, volume 4284 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006.

[53] R. D. Dean. *Formal Aspects of Mobile Code Security*. PhD thesis, Princeton University, January 1999.

[54] B. den Boer and A. Bosselaers. An Attack on the Last Two Rounds of MD4. In J. Feigenbaum, editor, *Advances in Cryptology – CRYPTO '91, Proceedings*, volume 576 of *Lecture Notes in Computer Science*, pages 194–203. Springer, 1992.

[55] B. den Boer and A. Bosselaers. Collisions for the Compression Function of MD5. In T. Helleseth, editor, *Advances in Cryptology – EUROCRYPT '93, Proceedings*, volume 765 of *Lecture Notes in Computer Science*, pages 293–304. Springer, 1994.

[56] P. Diaconis and F. Mosteller. Methods for Studying Coincidences. *Journal of the American Statistical Association*, 84(408):853–861, 1989.

[57] T. Dierks and C. Allen. The TLS Protocol Version 1.0. Internet Request for Comments (RFC) 2246, January 1999.

[58] W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[59] H. Dobbertin. Cryptanalysis of MD4. In D. Gollmann, editor, *Fast Software Encryption 1996, Proceedings*, volume 1039 of *Lecture Notes in Computer Science*, pages 53–69. Springer, 1996.

[60] H. Dobbertin. The Status of MD5 After a Recent Attack. *CryptoBytes*, 2(2):1–6, 1996.

[61] H. Dobbertin. Cryptanalysis of MD4. *Journal of Cryptology*, 11(4):253–271, 1998.

[62] H. Dobbertin. The First Two Rounds of MD4 are Not One-Way. In S. Vaudenay, editor, *Fast Software Encryption '98, Proceedings*, volume 1372 of *Lecture Notes in Computer Science*, pages 284–292. Springer, 1998.

[63] Y. Dodis, R. Gennaro, J. Håstad, H. Krawczyk, and T. Rabin. Randomness Extraction and Key Derivation Using the CBC, Cascade and HMAC Modes. In M. K. Franklin, editor, *Advances in Cryptology – CRYPTO 2004, Proceedings*, volume 3152 of *Lecture Notes in Computer Science*, pages 494–510. Springer, 2004.

[64] eBACS: ECRYPT Benchmarking of Cryptographic Systems. Measurements of hash functions. `http://bench.cr.yp.to/results-hash.html` (2008/11/21).

[65] The eSTREAM Project. `http://www.ecrypt.eu.org/stream/`.

[66] A. Evans Jr., W. Kantrowitz, and E. Weiss. A User Authentication Scheme Not Requiring Secrecy in the Computer. *Communications of the ACM*, 17(8):437–442, 1974.

[67] R. W. Floyd. Nondeterministic Algorithms. *Journal of the Association for Computing Machinery*, 14(4):636–644, 1967.

[68] P.-A. Fouque, J. Stern, and S. Zimmer. Cryptanalysis of Tweaked Versions of SMASH and Reparation. In *Selected Areas in Cryptography 2008, Proceedings*, Lecture Notes in Computer Science. Springer. To appear.

[69] P. Gauravaram and J. Kelsey. Linear-XOR and Additive Checksums Don't Protect Damgård-Merkle Hashes from Generic Attacks. In T. Malkin, editor, *Topics in Cryptology – CT-RSA 2008, Proceedings*, volume 4964 of *Lecture Notes in Computer Science*, pages 36–51. Springer, 2008.

[70] P. Gauravaram, J. Kelsey, L. R. Knudsen, and S. S. Thomsen. On hash functions using checksums. Technical Report MAT 2008-06, Department of Mathematics, Technical University of Denmark, November 2008. Available: `http://orbit.dtu.dk/getResource?recordId=228687&objectId=1&versionId=1` (2008/11/18).

[71] P. Gauravaram, L. R. Knudsen, K. Matusiewicz, F. Mendel, C. Rechberger, M. Schläffer, and S. S. Thomsen. `Grøstl` – a SHA-3 candidate. SHA-3 Algorithm Submission, October 31, 2008. Available: `http://www.groestl.info/Groestl.pdf` (2008/11/03).

[72] P. Gauravaram, W. Millan, E. Dawson, and K. Viswanathan. Constructing Secure Hash Functions by Enhancing Merkle-Damgård Construction. In L. M. Batten and R. Safavi-Naini, editors, *Australasian Conference on Information Security and Privacy (ACISP) 2006, Proceedings*, volume 4058 of *Lecture Notes in Computer Science*, pages 407–420. Springer, 2006.

[73] M. Gebhardt, G. Illies, and W. Schindler. A Note on the Practical Value of Single Hash Collisions for Special File Formats. In J. Dittmann, editor, *Sicherheit 2006, Beiträge der 3. Jahrestagung (Proceedings)*, volume 77 of *LNI*, pages 333–344. GI, 2006.

[74] H. Gilbert and H. Handschuh. Security Analysis of SHA-256 and Sisters. In M. Matsui and R. J. Zuccherato, editors, *Selected Areas in Cryptography 2003, Proceedings*, volume 3006 of *Lecture Notes in Computer Science*, pages 175–193. Springer, 2004.

[75] The GNU MP Bignum Library (2007). `http://gmplib.org` (2008/09/30).

[76] S. Goldwasser, S. Micali, and R. L. Rivest. A "Paradoxical" Solution To The Signature Problem (Extended Abstract). In *25th Annual Symposium on Foundations of Computer Science 1984, Proceedings*, pages 441–448. IEEE, 1984.

[77] Web page of the Grindahl hash functions. `http://www.ramkilde.com/grindahl`.

[78] Grøstl – a SHA-3 candidate. `http://www.groestl.info`.

[79] S. Gueron and M. E. Kounavis. Vortex: A New Family of One Way Hash Functions based on Rijndael Rounds and Carry-less Multiplication. Cryptology ePrint Archive, Report 2008/464, November 2008. `http://eprint.iacr.org/`.

[80] P. Hawkes, M. Paddon, and G. Rose. Automated Search for Round 1 Differentials for SHA-1: Work in Progress. Presented at *Second NIST Cryptographic Hash Workshop, August 24–25, 2006, Santa Barbara, California, USA.* Available: `http://csrc.nist.gov/groups/ST/hash/documents/HAWKES_sha1_obs_nist11%5B2%5D.pdf` (2008/11/25).

[81] S. Hirose. Provably Secure Double-Block-Length Hash Functions in a Black-Box Model. In C. Park and S. Chee, editors, *International Conference on Information Security and Cryptology (ICISC) 2004, Proceedings*, volume 3506 of *Lecture Notes in Computer Science*, pages 330–342. Springer, 2005.

[82] S. Hirose. Some Plausible Constructions of Double-Block-Length Hash Functions. In M. J. B. Robshaw, editor, *Fast Software Encryption 2006, Proceedings*, volume 4047 of *Lecture Notes in Computer Science*, pages 210–225. Springer, 2006.

[83] W. Hohl, X. Lai, T. Meier, and C. Waldvogel. Security of Iterated Hash Functions Based on Block Ciphers. In D. R. Stinson, editor,

Advances in Cryptology – CRYPTO '93, Proceedings, volume 773 of *Lecture Notes in Computer Science*, pages 379–390. Springer, 1994.

[84] S. Indesteege, F. Mendel, B. Preneel, and C. Rechberger. Collisions and other Non-Random Properties for Step-Reduced SHA-256. In R. Avanzi, editor, *Selected Areas in Cryptography 2008, Proceedings*, Lecture Notes in Computer Science. Springer. To appear.

[85] Intel Corporation. Advanced Encryption Standard (AES) Instructions Set. `http://softwarecommunity.intel.com/articles/eng/3788.htm`.

[86] International Organization for Standardization. ISO/IEC 10118-2:1994. Information technology – Security techniques – Hash-functions – Part 2: Hash-functions using an n-bit block cipher algorithm, 1994. Revised in 2000.

[87] International Organization for Standardization. ISO/IEC 10118-4:1998, Information technology – Security techniques – Hash-functions – Part 4: Hash-functions using modular arithmetic, 1998.

[88] International Organization for Standardization. ISO 9735-6:2002. Electronic data interchange for administration, commerce and transport (EDIFACT) – Application level syntax rules (Syntax version number: 4, Syntax release number: 1) – Part 6: Secure authentication and acknowledgement message (message type – AUTACK), 2002. Available: `http://www.gefeg.com/jswg/v41/data/V41-9735-6.pdf` (2008/09/02).

[89] International Organization for Standardization. ISO/IEC 18033-2:2006. Information technology – Security techniques – Encryption algorithms – Part 2: Asymmetric ciphers, 2006.

[90] A. Joux. Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions. In M. K. Franklin, editor, *Advances in Cryptology – CRYPTO 2004, Proceedings*, volume 3152 of *Lecture Notes in Computer Science*, pages 306–316. Springer, 2004.

[91] B. S. Kaliski Jr. The MD2 Message-Digest Algorithm. Internet Request for Comments (RFC) 1319, April 1992.

[92] B. S. Kaliski Jr. and M. Robshaw. Message Authentication with MD5. *CryptoBytes*, 1(1):5–8, 1995.

[93] J. Kelsey and T. Kohno. Herding Hash Functions and the Nostradamus Attack. In S. Vaudenay, editor, *Advances in Cryptology – EUROCRYPT 2006, Proceedings*, volume 4004 of *Lecture Notes in Computer Science*, pages 183–200. Springer, 2006.

[94] J. Kelsey and B. Schneier. Second Preimages on n-Bit Hash Functions for Much Less than 2^n Work. In R. Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005, Proceedings*, volume 3494 of *Lecture Notes in Computer Science*, pages 474–490. Springer, 2005.

[95] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. Internet Request for Comments (RFC) 2401, November 1998.

[96] A. Kipnis and A. Shamir. Cryptanalysis of the HFE Public Key Cryptosystem by Relinearization. In M. J. Wiener, editor, *Advances in Cryptology – CRYPTO '99, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 19–30. Springer, 1999.

[97] L. R. Knudsen. Truncated and Higher Order Differentials. In B. Preneel, editor, *Fast Software Encryption 1994, Proceedings*, volume 1008 of *Lecture Notes in Computer Science*, pages 196–211. Springer, 1995.

[98] L. R. Knudsen. SMASH – A Cryptographic Hash Function. In H. Gilbert and H. Handschuh, editors, *Fast Software Encryption 2005, Proceedings*, volume 3557 of *Lecture Notes in Computer Science*, pages 228–242. Springer, 2005.

[99] L. R. Knudsen and T. A. Berson. Truncated Differentials of SAFER. In D. Gollmann, editor, *Fast Software Encryption 1996, Proceedings*, volume 1039 of *Lecture Notes in Computer Science*, pages 15–26. Springer, 1996.

[100] L. R. Knudsen and X. Lai. New Attacks on all Double Block Length Hash Functions of Hash Rate 1, including the Parallel-DM. In A. De Santis, editor, *Advances in Cryptology – EUROCRYPT '94, Proceedings*, volume 950 of *Lecture Notes in Computer Science*, pages 410–418. Springer, 1995.

[101] L. R. Knudsen and J. E. Mathiassen. Preimage and Collision Attacks on MD2. In H. Gilbert and H. Handschuh, editors, *Fast Software Encryption 2005, Proceedings*, volume 3557 of *Lecture Notes in Computer Science*, pages 255–267. Springer, 2005.

[102] L. R. Knudsen, J. E. Mathiassen, F. Muller, and S. S. Thomsen. Cryptanalysis of MD2. Submitted to a journal, August 2007.

[103] L. R. Knudsen, F. Mendel, C. Rechberger, and S. S. Thomsen. Cryptanalysis of MDC-2. Submitted to an international conference, September 2008.

[104] L. R. Knudsen and B. Preneel. Hash Functions Based on Block Ciphers and Quaternary Codes. In K. Kim and T. Matsumoto, editors, *Advances in Cryptology – ASIACRYPT '96, Proceedings*, volume 1163 of *Lecture Notes in Computer Science*, pages 77–90. Springer, 1996.

[105] L. R. Knudsen and B. Preneel. Fast and Secure Hashing Based on Codes. In B. S. Kaliski Jr., editor, *Advances in Cryptology – CRYPTO '97, Proceedings*, volume 1294 of *Lecture Notes in Computer Science*, pages 485–498. Springer, 1997.

[106] L. R. Knudsen and B. Preneel. Construction of secure and fast hash functions using nonbinary error-correcting codes. *IEEE Transactions on Information Theory*, 48(9):2524–2539, 2002.

[107] L. R. Knudsen, C. Rechberger, and S. S. Thomsen. The Grindahl Hash Functions. In A. Biryukov, editor, *Fast Software Encryption 2007, Proceedings*, volume 4593 of *Lecture Notes in Computer Science*, pages 39–57. Springer, 2007.

[108] L. R. Knudsen and V. Rijmen. Known-Key Distinguishers for Some Block Ciphers. In K. Kurosawa, editor, *Advances in Cryptology – ASIACRYPT 2007, Proceedings*, volume 4833 of *Lecture Notes in Computer Science*, pages 315–324. Springer, 2007.

[109] L. R. Knudsen and S. S. Thomsen. Proposals for Iterated Hash Functions. In M. Malek, E. Fernández-Medina, and J. Hernando, editors, *SECRYPT 2006, Proceedings*, pages 246–253. INSTICC Press, 2006.

[110] L. R. Knudsen and S. S. Thomsen. Proposals for Iterated Hash Functions. In J. Filipe and M. S. Obaidat, editors, *E-Business and Telecommunication Networks. Third International Conference, ICETE 2006. Selected Papers.*, volume 9 of *Communications in Computer and Information Science*, pages 107–118. Springer, 2008.

[111] P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In N. Koblitz, editor, *Advances in*

Cryptology – CRYPTO '96, Proceedings, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.

[112] D. Kraus. Integrity mechanism in German and international payment systems, 2002. Available: `http://www.src-gmbh.de/whitepapers/Intergrity_mechanisms_in_payment_systems_Kraus_en.pdf` (2008/09/02).

[113] H. Krawczyk. On Extract-then-Expand Key Derivation Function and an HMAC-based KDF, March 2008. Manuscript. Available: `http://www.ee.technion.ac.il/~hugo/kdf/kdf.pdf` (2008/09/15).

[114] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. Internet Request for Comments (RFC) 2104, February 1997.

[115] X. Lai and J. L. Massey. Hash Functions Based on Block Ciphers. In R. A. Rueppel, editor, *Advances in Cryptology – EUROCRYPT '92, Proceedings*, volume 658 of *Lecture Notes in Computer Science*, pages 55–70. Springer, 1993.

[116] A. K. Lenstra and B. de Weger. On the Possibility of Constructing Meaningful Hash Collisions for Public Keys. In C. Boyd and J. M. G. Nieto, editors, *Australasian Conference on Information Security and Privacy (ACISP) 2005, Proceedings*, volume 3574 of *Lecture Notes in Computer Science*, pages 267–279. Springer, 2005.

[117] A. K. Lenstra, D. Page, and M. Stam. Discrete Logarithm Variants of VSH. In P. Q. Nguyen, editor, *Progress in Cryptology – VIETCRYPT 2006, Proceedings*, volume 4341 of *Lecture Notes in Computer Science*, pages 229–242. Springer, 2006.

[118] G. Leurent. Message Freedom in MD4 and MD5 Collisions: Application to APOP. In A. Biryukov, editor, *Fast Software Encryption 2007, Proceedings*, volume 4593 of *Lecture Notes in Computer Science*, pages 309–328. Springer, 2007.

[119] G. Leurent. MD4 is Not One-Way. In K. Nyberg, editor, *Fast Software Encryption 2008, Proceedings*, volume 5086 of *Lecture Notes in Computer Science*, pages 412–428. Springer, 2008.

[120] J. Linn. Privacy Enhancement for Internet Electronic Mail: Part III – Algorithms, Modes, and Identifiers. Internet Request for Comments (RFC) 1115, August 1989.

[121] S. Lucks. A Failure-Friendly Design Principle for Hash Functions. In B. K. Roy, editor, *Advances in Cryptology – ASIACRYPT 2005, Proceedings*, volume 3788 of *Lecture Notes in Computer Science*, pages 474–494. Springer, 2005.

[122] S. Lucks and M. Daum. The Story of Alice and her Boss: Hash Functions and the Blind Passenger Attack. Presented at the EUROCRYPT 2005 rump session. Available: `http://th.informatik.uni-mannheim.de/People/lucks/HashCollisions/` (2008/10/07).

[123] F. J. MacWilliams and N. J. A. Sloane. *The Theory of Error-Correcting Codes*. North-Holland Mathematical Library, 1977.

[124] W. Mao. *Modern Cryptography – Theory & Practice*. Prentice Hall, 2004.

[125] M. Matsui. Linear Cryptoanalysis Method for DES Cipher. In T. Helleseth, editor, *Advances in Cryptology – EUROCRYPT '93, Proceedings*, volume 765 of *Lecture Notes in Computer Science*, pages 386–397. Springer, 1994.

[126] M. Matsui. The First Experimental Cryptanalysis of the Data Encryption Standard. In Y. Desmedt, editor, *Advances in Cryptology – CRYPTO '94, Proceedings*, volume 839 of *Lecture Notes in Computer Science*, pages 1–11. Springer, 1994.

[127] S. M. Matyas, C. H. Meyer, and J. Oseas. Generating strong one-way functions with crypographic algorithm. *IBM Technical Disclosure Bulletin*, 27(10A):5658–5659, 1985.

[128] F. Mendel, N. Pramstaller, C. Rechberger, and V. Rijmen. Analysis of Step-Reduced SHA-256. In M. J. B. Robshaw, editor, *Fast Software Encryption 2006, Proceedings*, volume 4047 of *Lecture Notes in Computer Science*, pages 126–143. Springer, 2006.

[129] F. Mendel, N. Pramstaller, C. Rechberger, and V. Rijmen. The Impact of Carries on the Complexity of Collision Attacks on SHA-1. In M. J. B. Robshaw, editor, *Fast Software Encryption 2006, Proceedings*, volume 4047 of *Lecture Notes in Computer Science*, pages 278–292. Springer, 2006.

[130] F. Mendel and V. Rijmen. Weaknesses in the HAS-V Compression Function. In K.-H. Nam and G. Rhee, editors, *International Conference on Information Security and Cryptology (ICISC) 2007, Proceedings*,

volume 4817 of *Lecture Notes in Computer Science*, pages 335–345. Springer, 2007.

[131] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[132] R. C. Merkle. Secure Communications Over Insecure Channels. *Communications of the ACM*, 21(4):294–299, 1978.

[133] R. C. Merkle. A Fast Software One-Way Hash Function. *Journal of Cryptology*, 3(1):43–58, 1990.

[134] R. C. Merkle. One Way Hash Functions and DES. In G. Brassard, editor, *Advances in Cryptology – CRYPTO '89, Proceedings*, volume 435 of *Lecture Notes in Computer Science*, pages 428–446. Springer, 1990.

[135] C. H. Meyer and M. Schilling. Secure program load with manipulation detection code. In *SECURICOM 88, Proceedings*, pages 111–130, 1988.

[136] S. Miyaguchi, K. Ohta, and M. Iwata. 128-bit Hash function (N-Hash). *NTT Review*, 2(6):128–132, November 1990.

[137] F. Muller. The MD2 Hash Function Is Not One-Way. In P. J. Lee, editor, *Advances in Cryptology – ASIACRYPT 2004, Proceedings*, volume 3329 of *Lecture Notes in Computer Science*, pages 214–229. Springer, 2004.

[138] Y. Naito, Y. Sasaki, N. Kunihiro, and K. Ohta. Improved Collision Attack on MD4 with Probability Almost 1. In D. Won and S. Kim, editors, *International Conference on Information Security and Cryptology (ICISC) 2005, Proceedings*, volume 3935 of *Lecture Notes in Computer Science*, pages 129–145. Springer, 2006.

[139] Y. Naito, Y. Sasaki, T. Shimoyama, J. Yajima, N. Kunihiro, and K. Ohta. Improved Collision Search for SHA-0. In X. Lai and K. Chen, editors, *Advances in Cryptology – ASIACRYPT 2006, Proceedings*, volume 4284 of *Lecture Notes in Computer Science*, pages 21–36. Springer, 2006.

[140] National Bureau of Standards. Federal Information Processing Standards Publication (FIPS PUB) 46. Data Encryption Standard (DES), January 1977.

[141] National Institute of Standards and Technology. Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA-3) Family. In Federal Register Vol. 27, No. 212, November 2007.

[142] National Institute of Standards and Technology/U.S. Department of Commerce. Federal Information Processing Standards Publication (FIPS PUB) 180. Secure Hash Standard, May 1993.

[143] National Institute of Standards and Technology/U.S. Department of Commerce. Federal Information Processing Standards Publication (FIPS PUB) 180-1. Secure Hash Standard, April 1995.

[144] National Institute of Standards and Technology/U.S. Department of Commerce. Federal Information Processing Standards Publication (FIPS PUB) 186-2. Digital Signature Standard (DSS), January 2000.

[145] National Institute of Standards and Technology/U.S. Department of Commerce. Federal Information Processing Standards Publication (FIPS PUB) 197. Advanced Encryption Standard (AES), November 2001.

[146] National Institute of Standards and Technology/U.S. Department of Commerce. Federal Information Processing Standards Publication (FIPS PUB) 180-2. Secure Hash Standard, August 2002.

[147] National Institute of Standards and Technology/U.S. Department of Commerce. Federal Information Processing Standards Publication (FIPS PUB) 198. The Keyed-Hash Message Authentication Code (HMAC), March 2002.

[148] National Institute of Standards and Technology/U.S. Department of Commerce. Change Notice for FIPS PUB 180-2 [146], February 2004.

[149] National Institute of Standards and Technology/U.S. Department of Commerce. NIST Special Publication 800-56A. Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography (Revised), March 2007.

[150] National Institute of Standards and Technology/U.S. Department of Commerce. NIST Special Publication 800-57. Recommendation for Key Management – Part 1: General (Revised), March 2007.

[151] National Institute of Standards and Technology/U.S. Department of Commerce. NIST Special Publication 800-90. Recommendation for Random Number Generation Using Deterministic Random Bit Generators (Revised), March 2007.

[152] National Institute of Standards and Technology/U.S. Department of Commerce. NIST Special Publication 800-67. Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher (Revised), May 2008.

[153] D. A. Osvik. Speeding up Serpent. In *Third AES Candidate Conference*, pages 317–329, 2000. Available: `http://www.ii.uib.no/~osvik/pub/aes3.pdf` (2008/11/25).

[154] T. Peyrin. Cryptanalysis of Grindahl. In K. Kurosawa, editor, *Advances in Cryptology – ASIACRYPT 2007, Proceedings*, volume 4833 of *Lecture Notes in Computer Science*, pages 551–567. Springer, 2007.

[155] T. Peyrin, H. Gilbert, F. Muller, and M. J. B. Robshaw. Combining Compression Functions and Block Cipher-Based Hash Functions. In X. Lai and K. Chen, editors, *Advances in Cryptology – ASIACRYPT 2006, Proceedings*, volume 4284 of *Lecture Notes in Computer Science*, pages 315–331. Springer, 2006.

[156] J. M. Pollard. A Monte Carlo method for factorisation. *BIT*, 15(3):331–334, 1975.

[157] N. Pramstaller, C. Rechberger, and V. Rijmen. Preliminary Analysis of the SHA-256 Message Expansion. Presented at *NIST Cryptographic Hash Workshop, October 31–November 1, 2005, Gaithersburg, Maryland, USA.* Available: `http://csrc.nist.gov/groups/ST/hash/documents/Rechberger_PreliminaryAnalysisOfSHA256.pdf` (2008/11/25).

[158] N. Pramstaller, C. Rechberger, and V. Rijmen. Exploiting Coding Theory for Collision Attacks on SHA-1. In N. P. Smart, editor, *Cryptography and Coding 2005, Proceedings*, volume 3796 of *Lecture Notes in Computer Science*, pages 78–95. Springer, 2005.

[159] N. Pramstaller, C. Rechberger, and V. Rijmen. Impact of Rotations in SHA-1 and Related Hash Functions. In B. Preneel and S. E. Tavares, editors, *Selected Areas in Cryptography 2005, Proceedings*, volume 3897 of *Lecture Notes in Computer Science*, pages 261–275. Springer, 2006.

[160] B. Preneel. *Analysis and Design of Cryptographic Hash Functions*. PhD thesis, Katholieke Universiteit Leuven, February 1993.

[161] B. Preneel, A. Bosselaers, R. Govaerts, and J. Vandewalle. Collision-free hashfunctions based on blockcipher algorithms. In *International Carnahan Conference on Security Technology 1989, Proceedings*, pages 203–210. IEEE, 1989.

[162] B. Preneel, R. Govaerts, and J. Vandewalle. On the Power of Memory in the Design of Collision Resistant Hash Functions. In J. Seberry and Y. Zheng, editors, *Advances in Cryptology – ASIACRYPT '92, Proceedings*, volume 718 of *Lecture Notes in Computer Science*, pages 105–121. Springer, 1993.

[163] B. Preneel, R. Govaerts, and J. Vandewalle. Hash Functions Based on Block Ciphers: A Synthetic Approach. In D. R. Stinson, editor, *Advances in Cryptology – CRYPTO '93, Proceedings*, volume 773 of *Lecture Notes in Computer Science*, pages 368–378. Springer, 1994.

[164] B. Preneel and P. C. van Oorschot. MDx-MAC and Building Fast MACs from Hash Functions. In D. Coppersmith, editor, *Advances in Cryptology – CRYPTO '95, Proceedings*, volume 963 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 1995.

[165] G. B. Purdy. A High Security Log-in Procedure. *Communications of the ACM*, 17(8):442–445, 1974.

[166] J.-J. Quisquater and M. Girault. 2n-Bit Hash-Functions Using n-Bit Symmetric Block Cipher Algorithms. In J.-J. Quisquater and J. Vandewalle, editors, *Advances in Cryptology – EUROCRYPT '89, Proceedings*, volume 434 of *Lecture Notes in Computer Science*, pages 102–109. Springer, 1990.

[167] M. O. Rabin. Digitalized signatures. In R. Lipton and R. DeMillo, editors, *Foundations of Secure Computation*, pages 155–166. Academic Press, 1978.

[168] M. O. Rabin. Digitalized Signatures and Public Key Functions as Intractable as Factorization. Technical Report 212, MIT, 1979. Available: `http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TR-212.pdf` (2008/09/17).

[169] V. Rijmen and E. Oswald. Update on SHA-1. In A. Menezes, editor, *Topics in Cryptology – CT-RSA 2005, Proceedings*, volume 3376 of *Lecture Notes in Computer Science*, pages 58–71. Springer, 2005.

[170] R. L. Rivest. The MD4 Message Digest Algorithm. Internet Request for Comments (RFC) 1186, October 1990.

[171] R. L. Rivest. The MD4 Message Digest Algorithm. In A. Menezes and S. A. Vanstone, editors, *Advances in Cryptology – CRYPTO '90, Proceedings*, volume 537 of *Lecture Notes in Computer Science*, pages 303–311. Springer, 1991.

[172] R. L. Rivest. The MD4 Message-Digest Algorithm. Internet Request for Comments (RFC) 1320, April 1992.

[173] R. L. Rivest. The MD5 Message-Digest Algorithm. Internet Request for Comments (RFC) 1321, April 1992.

[174] R. L. Rivest, A. Shamir, and L. M. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[175] P. Rogaway. Formalizing Human Ignorance. In P. Q. Nguyen, editor, *Progress in Cryptology – VIETCRYPT 2006, Proceedings*, volume 4341 of *Lecture Notes in Computer Science*, pages 211–228. Springer, 2006.

[176] P. Rogaway and T. Shrimpton. Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance. In B. K. Roy and W. Meier, editors, *Fast Software Encryption 2004, Proceedings*, volume 3017 of *Lecture Notes in Computer Science*, pages 371–388. Springer, 2004.

[177] P. Rogaway and J. Steinberger. Security/Efficiency Tradeoffs for Permutation-Based Hashing. In N. Smart, editor, *Advances in Cryptology – EUROCRYPT 2008, Proceedings*, volume 4965 of *Lecture Notes in Computer Science*, pages 220–236. Springer, 2008.

[178] P. Rogaway and J. P. Steinberger. Constructing Cryptographic Hash Functions from Fixed-Key Blockciphers. In D. Wagner, editor, *Advances in Cryptology – CRYPTO 2008, Proceedings*, volume 5157 of *Lecture Notes in Computer Science*, pages 433–450. Springer, 2008.

[179] N. Rogier and P. Chauvaud. MD2 Is not Secure without the Checksum Byte. *Designs, Codes and Cryptography*, 12(3):245–251, 1997.

[180] Rostekhregulirovaniye (Russia's Federal Agency for Technical Regulation and Metrology). GOST R 34.11-94: Information technology – Cryptographic data security – Hashing function, 1994.

[181] RSA Laboratories. PKCS #1: RSA Cryptography Standard (Version 2.1, June 14, 2002). Available: `http://www.rsa.com/rsalabs/node.asp?id=2125` (2008/09/30).

[182] RSA Laboratories. PKCS #5: Password-Based Cryptography Standard (Version 2.0, March 25, 1999). Available: `http://www.rsa.com/rsalabs/node.asp?id=2127` (2008/09/30).

[183] M.-J. O. Saarinen. Security of VSH in the Real World. In R. Barua and T. Lange, editors, *Progress in Cryptology – INDOCRYPT 2006, Proceedings*, volume 4329 of *Lecture Notes in Computer Science*, pages 95–103. Springer, 2006.

[184] S. K. Sanadhya and P. Sarkar. New Local Collisions for the SHA-2 Hash Family. In K.-H. Nam and G. Rhee, editors, *International Conference on Information Security and Cryptology (ICISC) 2007, Proceedings*, volume 4817 of *Lecture Notes in Computer Science*, pages 193–205. Springer, 2007.

[185] S. K. Sanadhya and P. Sarkar. Attacking Reduced Round SHA-256. In S. M. Bellovin, R. Gennaro, A. D. Keromytis, and M. Yung, editors, *Applied Cryptography and Network Security (ACNS) 2008, Proceedings*, volume 5037 of *Lecture Notes in Computer Science*, pages 130–143. Springer, 2008.

[186] S. K. Sanadhya and P. Sarkar. Non-linear Reduced Round Attacks against SHA-2 Hash Family. In Y. Mu, W. Susilo, and J. Seberry, editors, *Australasian Conference on Information Security and Privacy (ACISP) 2008, Proceedings*, volume 5107 of *Lecture Notes in Computer Science*, pages 254–266. Springer, 2008.

[187] Y. Sasaki and K. Aoki. Preimage Attacks on Step-Reduced MD5. In Y. Mu, W. Susilo, and J. Seberry, editors, *Australasian Conference on Information Security and Privacy (ACISP) 2008, Proceedings*, volume 5107 of *Lecture Notes in Computer Science*, pages 282–296. Springer, 2008.

[188] Y. Sasaki, Y. Naito, N. Kunihiro, and K. Ohta. Improved Collision Attacks on MD4 and MD5. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E90-A(1):36–47, 2007.

[189] Y. Sasaki, L. Wang, N. Kunihiro, and K. Ohta. New Message Differences for Collision Attacks on MD4 and MD5. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E91-A(1):55–63, 2008.

[190] Y. Sasaki, L. Wang, K. Ohta, and N. Kunihiro. New Message Difference for MD4. In A. Biryukov, editor, *Fast Software Encryption 2007, Proceedings*, volume 4593 of *Lecture Notes in Computer Science*, pages 329–348. Springer, 2007.

[191] Y. Sasaki, L. Wang, K. Ohta, and N. Kunihiro. Security of MD5 Challenge and Response: Extension of APOP Password Recovery Attack. In T. Malkin, editor, *Topics in Cryptology – CT-RSA 2008, Proceedings*, volume 4964 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2008.

[192] M. Schläffer and E. Oswald. Searching for Differential Paths in MD4. In M. J. B. Robshaw, editor, *Fast Software Encryption 2006, Proceedings*, volume 4047 of *Lecture Notes in Computer Science*, pages 242–261. Springer, 2006.

[193] Quote: Bruce Schneier, NIST Cryptographic Hash Workshop, October 31–November 1, 2005, Gaithersburg, Maryland, USA.

[194] The SHA-3 Zoo. `http://ehash.iaik.tugraz.at/wiki/The_SHA-3_Zoo`.

[195] T. Shrimpton and M. Stam. Building a Collision-Resistant Compression Function from Non-compressing Primitives. In L. Aceto, I. Damgård, L. A. Goldberg, M. M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz, editors, *International Colloquium on Automata, Languages and Programming (ICALP) 2008, Proceedings*, volume 5126 of *Lecture Notes in Computer Science*, pages 643–654. Springer, 2008.

[196] M. Stam. Beyond Uniformity: Better Security/Efficiency Tradeoffs for Compression Functions. In D. Wagner, editor, *Advances in Cryptology – CRYPTO 2008, Proceedings*, volume 5157 of *Lecture Notes in Computer Science*, pages 397–412. Springer, 2008.

[197] J. P. Steinberger. The Collision Intractability of MDC-2 in the Ideal-Cipher Model. In M. Naor, editor, *Advances in Cryptology – EUROCRYPT 2007, Proceedings*, volume 4515 of *Lecture Notes in Computer Science*, pages 34–51. Springer, 2007.

[198] M. Stevens, A. K. Lenstra, and B. de Weger. Chosen-Prefix Collisions for MD5 and Colliding X.509 Certificates for Different Identities. In M. Naor, editor, *Advances in Cryptology – EUROCRYPT 2007, Proceedings*, volume 4515 of *Lecture Notes in Computer Science*, pages 1–22. Springer, 2007.

[199] M. M. J. Stevens. On Collisions for MD5. Master's thesis, Technische Universiteit Eindhoven/Eindhoven University of Technology, June 2007. Available: `http://www.win.tue.nl/hashclash/On%20Collisions%20for%20MD5%20-%20M.M.J.%20Stevens.pdf` (2008/09/25).

[200] B. Struif. German Health Professional Card and Security Module Card, Specification, Pharmacist & Physician, v. 2.0, 2003. Available: `http://www.dkgev.de/media/file/2589.spez-engl-3.pdf` (2008/09/02).

[201] M. Sugita, M. Kawazoe, L. Perret, and H. Imai. Algebraic Cryptanalysis of 58-Round SHA-1. In A. Biryukov, editor, *Fast Software Encryption 2007, Proceedings*, volume 4593 of *Lecture Notes in Computer Science*, pages 349–365. Springer, 2007.

[202] K. Suzuki, D. Tonien, K. Kurosawa, and K. Toyota. Birthday Paradox for Multi-collisions. In M. S. Rhee and B. Lee, editors, *International Conference on Information Security and Cryptology (ICISC) 2006, Proceedings*, volume 4296 of *Lecture Notes in Computer Science*, pages 29–40. Springer, 2006.

[203] S. S. Thomsen. Website of Søren Steffen Thomsen. `http://www.mat.dtu.dk/people/S.Thomsen`.

[204] S. S. Thomsen. Website of the ANACONDA hash functions. `http://www.mat.dtu.dk/people/S.Thomsen/anaconda`.

[205] S. S. Thomsen. Cryptographic Hash Functions. Master's thesis, Danmarks Tekniske Universitet/Technical University of Denmark, November 2005. Available: `http://www.mat.dtu.dk/people/S.Thomsen/sst_masters.pdf` (2008/09/25).

[206] S. S. Thomsen. The ANACONDA hash functions. Technical Report MAT 2008-05, Department of Mathematics, Technical University of Denmark, November 2008. Available: `http://orbit.dtu.dk/getResource?recordId=228495&objectId=1&versionId=1` (2008/11/12).

[207] G. Tsudik. Message Authentication with One-Way Hash Functions. In *INFOCOM '92, Proceedings*, pages 2055–2059, 1992.

[208] P. C. van Oorschot and M. J. Wiener. Parallel Collision Search with Application to Hash Functions and Discrete Logarithms. In *ACM Conference on Computer and Communications Security 1994, Proceedings*, pages 210–218. ACM, 1994.

[209] P. C. van Oorschot and M. J. Wiener. Parallel Collision Search with Cryptanalytic Applications. *Journal of Cryptology*, 12(1):1–28, 1999.

[210] H. C. A. van Tilborg, editor. *Encyclopedia of Cryptography and Security*. Springer, 2005.

[211] S. Vaudenay. On the Need for Multipermutations: Cryptanalysis of MD4 and SAFER. In B. Preneel, editor, *Fast Software Encryption 1994, Proceedings*, volume 1008 of *Lecture Notes in Computer Science*, pages 286–297. Springer, 1995.

[212] J. Viega. The AHASH Mode of Operation, September 2004. Manuscript. Available: `http://www.cryptobarn.com/papers/ahash.pdf` (2008/09/02).

[213] D. Wagner. A Generalized Birthday Problem. In M. Yung, editor, *Advances in Cryptology – CRYPTO 2002, Proceedings*, volume 2442 of *Lecture Notes in Computer Science*, pages 288–303. Springer, 2002.

[214] X. Wang, X. Lai, D. Feng, H. Chen, and X. Yu. Cryptanalysis of the Hash Functions MD4 and RIPEMD. In R. Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005, Proceedings*, volume 3494 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2005.

[215] X. Wang, Y. L. Yin, and H. Yu. Finding Collisions in the Full SHA-1. In V. Shoup, editor, *Advances in Cryptology – CRYPTO 2005, Proceedings*, volume 3621 of *Lecture Notes in Computer Science*, pages 17–36. Springer, 2005.

[216] X. Wang and H. Yu. How to Break MD5 and Other Hash Functions. In R. Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005, Proceedings*, volume 3494 of *Lecture Notes in Computer Science*, pages 19–35. Springer, 2005.

[217] X. Wang, H. Yu, and Y. L. Yin. Efficient Collision Search Attacks on SHA-0. In V. Shoup, editor, *Advances in Cryptology – CRYPTO 2005, Proceedings*, volume 3621 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2005.

[218] D. Watanabe. A note on the security proof of Knudsen-Preneel construction of a hash function, 2006. Manuscript. Available: `http://csrc.nist.gov/groups/ST/hash/documents/WATANABE_kp_attack.pdf` (2008/07/18).

[219] M. V. Wilkes. *Time-Sharing Computer Systems*. Macdonald and Jane's, 1968.

[220] R. S. Winternitz. A Secure One-Way Hash Function Built from DES. In *IEEE Symposium on Security and Privacy*, pages 88–90, 1984.

[221] J. Yajima, Y. Sasaki, Y. Naito, T. Iwasaki, T. Shimoyama, N. Kunihiro, and K. Ohta. A New Strategy for Finding a Differential Path of SHA-1. In J. Pieprzyk, H. Ghodosi, and E. Dawson, editors, *Australasian Conference on Information Security and Privacy (ACISP) 2007, Proceedings*, volume 4586 of *Lecture Notes in Computer Science*, pages 45–58. Springer, 2007.

[222] H. Yoshida and A. Biryukov. Analysis of a SHA-256 Variant. In B. Preneel and S. E. Tavares, editors, *Selected Areas in Cryptography 2005, Proceedings*, volume 3897 of *Lecture Notes in Computer Science*, pages 245–260. Springer, 2006.

[223] H. Yu and X. Wang. Multi-collision Attack on the Compression Functions of MD4 and 3-Pass HAVAL. In K.-H. Nam and G. Rhee, editors, *International Conference on Information Security and Cryptology (ICISC) 2007, Proceedings*, volume 4817 of *Lecture Notes in Computer Science*, pages 206–226. Springer, 2007.

[224] G. Yuval. How to swindle Rabin. *Cryptologia*, 3(3):187–189, 1979.

[225] Y. Zheng, T. Matsumoto, and H. Imai. Connections among Several Versions of One-Way Hash Functions. *IEICE Transactions (1976–1990). Special Issue on Cryptography and Information Security*, E73-E(7):1092–1099, 1990.

[226] Y. Zheng, T. Matsumoto, and H. Imai. Structural Properties of One-way Hash Functions. In A. Menezes and S. A. Vanstone, editors, *Advances in Cryptology – CRYPTO '90, Proceedings*, volume 537 of *Lecture Notes in Computer Science*, pages 285–302. Springer, 1991.

[227] Y. Zheng, J. Pieprzyk, and J. Seberry. HAVAL – A One-Way Hashing Algorithm with Variable Length of Output. In J. Seberry and Y. Zheng, editors, *Advances in Cryptology – ASIACRYPT '92, Proceedings*, volume 718 of *Lecture Notes in Computer Science*, pages 83–104. Springer, 1993.

Index